

COMO O LINUX FUNCIONA

O QUE TODO SUPERUSUÁRIO DEVERIA SABER

BRIAN WARD

novatec





ELOGIOS À PRIMEIRA EDIÇÃO DE *COMO O LINUX FUNCIONA*

“ Um ótimo recurso. Em aproximadamente 440 páginas, o livro discute tudo sobre o básico.”

– EWEEK

“ Definitivamente, recomendo este livro àqueles que estiverem interessados no Linux, porém não tiveram experiência para conhecer o funcionamento interno desse sistema operacional.”

– O'REILLYNET

“ Um dos melhores livros básicos para conhecer o Linux, escrito com vistas ao usuário capacitado. Cinco estrelas.”

– OPENSOURCE-BOOK-REVIEWS.COM

“ Admiravelmente bem-sucedido por causa da maneira como está organizado e pelo nível dos detalhes técnicos oferecidos.”

– KICKSTART NEWS

“ Essa é uma introdução bem diferente ao Linux. É despretensioso, concentra-se na linha de comando e explora o funcionamento interno em vez de focar em frontends GUI que se apresentam como ferramentas MS Windows mais familiares.”

– TECHBOOKREPORT.COM

“ Este livro faz um ótimo trabalho ao explicar as engrenagens envolvidas no funcionamento do Linux.”

– HOSTING RESOLVE

COMO O LINUX FUNCIONA

O QUE TODO SUPERUSUÁRIO DEVERIA SABER

Brian Ward

Novatec

Copyright © 2014 by Brian Ward. Title of English-language original: *How Linux Works, 2nd Edition*, ISBN 978-1-59327-567-9, published by No Starch Press. Portuguese-language edition copyright © 2015 by Novatec Editora Ltda. All rights reserved.

Copyright © 2014 por Brian Ward. Título original em inglês: *How Linux Works, 2nd Edition*, ISBN 978-1-59327-567-9, publicado pela No Starch Press. Edição em português copyright © 2015 pela Novatec Editora Ltda. Todos os direitos reservados.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998.

É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Tradução: Lúcia A. Kinoshita

Revisão gramatical: Marta Almeida de Sá

Editoração eletrônica: Carolina Kuwabata

Assistente editorial: Priscila A. Yoshimatsu

ISBN: 978-85-7522-578-3

Histórico de edições impressas:

Outubro/2016 Primeira reimpressão

Fevereiro/2015 Primeira edição

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

Email: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

Sumário

Prefácio

- Quem deve ler este livro?
- Pré-requisitos
- Como ler este livro
- Uma abordagem prática
- Como este livro está organizado
- Quais são as novidades na segunda edição? Uma observação sobre a terminologia

Agradecimentos

Capítulo 1 ■ Quadro geral

- 1.1 Níveis e camadas de abstração em um sistema Linux
- 1.2 Hardware: compreendendo a memória principal
- 1.3 Kernel
- 1.4 Espaço de usuário
- 1.5 Usuários
- 1.6 Próximos passos

Capítulo 2 ■ Comandos básicos e hierarquia de diretórios

- 2.1 Bourne shell: /bin/sh
- 2.2 Usando o shell
- 2.3 Comandos básicos
- 2.4 Navegando pelos diretórios
- 2.5 Comandos intermediários
- 2.6 Alterando sua senha e o shell
- 2.7 Arquivos ponto
- 2.8 Variáveis de ambiente e de shell
- 2.9 O path do comando
- 2.10 Caracteres especiais
- 2.11 Edição de linha de comando
- 2.12 Editores de texto
- 2.13 Obtendo ajuda online
- 2.14 Entrada e saída de shell

- 2.15 Entendendo as mensagens de erro
- 2.16 Listando e manipulando processos
- 2.17 Modos e permissões de arquivo
- 2.18 Arquivamento e compressão de arquivos
- 2.19 Aspectos essenciais da hierarquia de diretórios do Linux
- 2.20 Executando comandos como superusuário
- 2.21 Próximos passos

Capítulo 3 ■ Dispositivos

- 3.1 Arquivos de dispositivo
- 3.2 Path de dispositivos sysfs
- 3.3 dd e dispositivos
- 3.4 Resumo dos nomes dos dispositivos
- 3.5 udev
- 3.6 Em detalhes: SCSI e o kernel do Linux

Capítulo 4 ■ Discos e sistemas de arquivos

- 4.1 Particionando dispositivos de disco
- 4.2 Sistemas de arquivos
- 4.3 Espaço de swap
- 4.4 Próximos passos: discos e espaço de usuário
- 4.5 Dentro de um sistema de arquivos tradicional

Capítulo 5 ■ Como o kernel do Linux inicializa

- 5.1 Mensagens de inicialização
- 5.2 Inicialização do kernel e opções de boot
- 5.3 Parâmetros do kernel
- 5.4 Boot loaders
- 5.5 Introdução ao GRUB
- 5.6 Problemas com o boot seguro do UEFI
- 5.7 Carregando outros sistemas operacionais em cadeia
- 5.8 Detalhes dos boot loaders

Capítulo 6 ■ Como o espaço de usuário inicia

- 6.1 Introdução ao init
- 6.2 Os runlevels do System V
- 6.3 Identificando o seu init
- 6.4 systemd
- 6.5 Upstart

6.6 System V init

6.7 Desligando o seu sistema

6.8 O sistema de arquivos inicial em RAM

6.9 Inicialização de emergência e modo monousuário

Capítulo 7 ■ Configuração do sistema: logging, hora do sistema, tarefas em lote e usuários

7.1 Estrutura do /etc

7.2 Logging do sistema

7.3 Arquivos para gerenciamento de usuários

7.4 getty e login

7.5 Configurando o horário

7.6 Agendando tarefas recorrentes com cron

7.7 Agendando tarefas a serem executadas uma só vez com at

7.8 Entendendo os IDs de usuário e as mudanças de usuário

7.9 Identificação e autenticação de usuários

7.10 PAM

7.11 Próximos passos

Capítulo 8 ■ Observando mais de perto os processos e a utilização de recursos

8.1 Monitorando processos

8.2 Encontrando arquivos abertos com lsof

8.3 Execução de programas de tracing e chamadas de sistema

8.4 Threads

8.5 Introdução à monitoração de recursos

8.6 Medindo o tempo de CPU

8.7 Ajustando as prioridades do processo

8.8 Médias de carga

8.9 Memória

8.10 Monitorando o desempenho da CPU e da memória com vmstat

8.11 Monitoração de I/O

8.12 Monitoração por processo com pidstat

8.13 Tópicos adicionais

Capítulo 9 ■ Entendendo a rede e sua configuração

9.1 Básico sobre redes

9.2 Camadas de rede

9.3 A camada de Internet

- 9.4 Rotas e a tabela de roteamento do kernel
- 9.5 Ferramentas básicas para ICMP e DNS
- 9.6 A camada física e a Ethernet
- 9.7 Entendendo as interfaces de rede do kernel
- 9.8 Introdução à configuração das interfaces de rede
- 9.9 Configuração de rede ativada no boot
- 9.10 Problemas com a configuração manual de rede ativada no boot
- 9.11 Gerenciadores de configuração de rede
- 9.12 Resolvendo nomes de hosts
- 9.13 Localhost
- 9.14 A camada de transporte: TCP, UDP e serviços
- 9.15 Analisando uma rede local simples novamente
- 9.16 Entendendo o DHCP
- 9.17 Configurando o Linux como roteador
- 9.18 Redes privadas
- 9.19 Tradução de endereços de rede (Mascaramento de IP)
- 9.20 Roteadores e o Linux
- 9.21 Firewalls
- 9.22 Ethernet, IP e ARP
- 9.23 Ethernet wireless
- 9.24 Resumo

Capítulo 10 ■ Aplicações e serviços de rede

- 10.1 Básico sobre serviços
- 10.2 Servidores de rede
- 10.3 Secure Shell (SSH)
- 10.4 Os daemons inetd e xinetd
- 10.5 Ferramentas de diagnóstico
- 10.6 Remote Procedure Call (RPC)
- 10.7 Segurança em redes
- 10.8 Próximos passos
- 10.9 Sockets: como os processos se comunicam com a rede
- 10.10 Sockets de domínio Unix

Capítulo 11 ■ Introdução aos shell scripts

- 11.1 Básico sobre shell scripts
- 11.2 Uso de aspas e literais
- 11.3 Variáveis especiais
- 11.4 Códigos de saída

- 11.5 Condicionais
- 11.6 Laços
- 11.7 Substituição de comandos
- 11.8 Gerenciamento de arquivos temporários
- 11.9 Here documents
- 11.10 Utilitários importantes para shell scripts
- 11.11 Subshells
- 11.12 Incluindo outros arquivos em scripts
- 11.13 Lendo dados de entrada do usuário
- 11.14 Quando (não) usar shell scripts

Capítulo 12 ■ Movendo arquivos pela rede

- 12.1 Cópia rápida
- 12.2 rsync
- 12.3 Introdução ao compartilhamento de arquivos
- 12.4 Compartilhando arquivos com o Samba
- 12.5 Clientes NFS
- 12.6 Outras opções e limitações do serviço de arquivos em redes

Capítulo 13 ■ Ambientes de usuário

- 13.1 Diretrizes para criar arquivos de inicialização
- 13.2 Quando os arquivos de inicialização devem ser alterados
- 13.3 Elementos do arquivo de inicialização do shell
- 13.4 Ordem dos arquivos de inicialização e exemplos
- 13.5 Configurações default do usuário
- 13.6 Armadilhas dos arquivos de inicialização
- 13.7 Tópicos adicionais relativos à inicialização

Capítulo 14 ■ Uma breve análise do desktop Linux

- 14.1 Componentes do desktop
- 14.2 Um olhar mais detalhado sobre o X Window System
- 14.3 Explorando os clientes do X
- 14.4 O futuro do X
- 14.5 D-Bus
- 14.6 Impressão
- 14.7 Outros assuntos relacionados ao desktop

Capítulo 15 ■ Ferramentas de desenvolvimento

- 15.1 O compilador C

- 15.2 make
- 15.3 Debuggers
- 15.4 Lex e Yacc
- 15.5 Linguagens de scripting
- 15.6 Java
- 15.7 Próximos passos: compilando pacotes

Capítulo 16 ■ Introdução à compilação de software a partir de código-fonte C

- 16.1 Sistemas de geração de software
- 16.2 Desempacotando pacotes com código-fonte C
- 16.3 GNU autoconf
- 16.4 Efetuando instalações
- 16.5 Aplicando um patch
- 16.6 Resolvendo problemas de compilação e de instalações
- 16.7 Próximos passos

Capítulo 17 ■ Desenvolvendo sobre o básico

- 17.1 Servidores e aplicações web
- 17.2 Bancos de dados
- 17.3 Virtualização
- 17.4 Processamento distribuído e por demanda
- 17.5 Sistemas embarcados
- 17.6 Observações finais

Bibliografia

Prefácio

Escrevi este livro porque acredito que você deva ter a oportunidade de saber o que o seu computador faz. Você deve ser capaz de fazer o seu software se comportar da maneira que você quiser (dentro dos limites razoáveis das capacidades desse software, é claro). O segredo para conquistar esse poder está em compreender os fundamentos daquilo que o software faz e como ele funciona, e esse é o assunto deste livro. Você jamais deverá lutar contra um computador.

O Linux é uma ótima plataforma de aprendizado porque ele não tenta esconder nada de você. Em particular, quase toda a configuração do sistema pode ser encontrada em arquivos-textos simples, que são bem fáceis de ler. A única parte complicada é descobrir quais porções são responsáveis por quais tarefas e como tudo se encaixa.

Quem deve ler este livro?

Seu interesse em conhecer o funcionamento do Linux pode resultar de diversos motivos. No campo profissional, o pessoal de operações e de DevOps (Development and Operations, ou Desenvolvimento e operações) deve saber praticamente tudo o que se encontra neste livro. Arquitetos e desenvolvedores de software para Linux também devem conhecer este material para que possam fazer o melhor uso possível do sistema operacional. Pesquisadores e estudantes, que normalmente executam seus próprios sistemas Linux, também encontrarão explicações úteis neste livro, mostrando *por que* as configurações foram feitas do modo como estão.

E há também os curiosos – pessoas que simplesmente amam trabalhar com seus computadores por diversão, lucro ou ambos os motivos. Quer saber por que certas tarefas funcionam e outras não? Quer saber o que acontecerá se você mover algo de lugar? Provavelmente, você é um curioso.

Pré-requisitos

Embora o Linux seja amado pelos programadores, não é necessário ser um para ler este livro: você só precisa ter conhecimentos básicos de um usuário de computador. Isso quer dizer que você deve ser capaz de lidar com uma GUI (especialmente com o instalador e a interface de configuração de uma distribuição Linux) e saber o que são

arquivos e diretórios (pastas). Você também deverá estar preparado para consultar documentações adicionais em seu sistema e na Web. Conforme mencionado anteriormente, o mais importante será estar pronto e disposto a trabalhar com o seu computador.

Como ler este livro

Reunir os conhecimentos necessários é um desafio para lidar com qualquer assunto técnico. Ao explicar como os sistemas de software funcionam, a situação pode *realmente* ficar complicada. Detalhes demais desanimarão o leitor e farão com que seja difícil aprender as atividades importantes (o cérebro humano simplesmente não é capaz de processar tantos conceitos novos de uma só vez), porém menos detalhes do que o necessário deixarão o leitor no escuro e despreparado para assuntos que venham a seguir.

A maioria dos capítulos foi concebida de modo a lidar com os assuntos mais importantes antes, ou seja, as informações básicas necessárias para progredir. Em determinados lugares, fiz simplificações para manter o foco. À medida que um capítulo avançar, haverá muito mais detalhes, especialmente nas últimas seções. Você precisa conhecer essas partes imediatamente? Na maioria dos casos, não, como indico normalmente. Se os seus olhos começarem a ficar embaçados ao encarar muitos detalhes extras sobre assuntos que você acabou de aprender, não hesite em pular para o próximo capítulo ou simplesmente faça uma pausa. Os pequenos detalhes continuarão lá, esperando por você.

Uma abordagem prática

Independentemente da maneira escolhida para usar este livro, você deverá estar diante de um computador Linux, de preferência um do qual você possa abusar de forma segura ao fazer seus experimentos. Pode ser que você prefira trabalhar com uma instalação virtual – eu usei o VirtualBox para testar boa parte do material contido neste livro. Você deverá ter acesso de superusuário (root), porém uma conta normal de usuário será usada na maioria das vezes. Em boa parte do tempo, você trabalhará na linha de comando, em uma janela de terminal ou em um sessão remota. Se você ainda não trabalhou muito nesse ambiente, tudo bem; o capítulo 2 irá prepará-lo.

Os comandos neste livro normalmente terão a seguinte aparência:

```
$ ls /  
[alguma saída]
```

Insira o texto que estiver em negrito; o texto que se segue – que não está em negrito – é o que o computador devolverá como resposta. O \$ é o prompt para a sua conta normal de usuário. Se houver um # como prompt, você deverá ser um superusuário. (Mais sobre esse assunto no capítulo 2.)

Como este livro está organizado

Os capítulos do livro foram agrupados em três partes básicas. A primeira parte é introdutória; ela oferece uma visão geral do sistema, além de permitir uma experiência prática com algumas ferramentas que serão necessárias enquanto você estiver executando o Linux. Em seguida, cada parte do sistema será explorada com mais detalhes, do gerenciamento de dispositivos à configuração de rede, seguindo a ordem geral da inicialização do sistema. Por fim, você fará um tour por algumas partes de um sistema em execução, conhecerá algumas habilidades essenciais e obterá informações sobre as ferramentas usadas pelos programadores.

Com exceção do capítulo 2, a maior parte dos capítulos iniciais envolve intensamente o kernel do Linux, porém você conhecerá o espaço de usuário à medida que o livro avançar. (Se não souber sobre o que estou falando aqui, não se preocupe; isso será explicado no capítulo 1.)

O material deste livro foi concebido para ser o mais independente possível de distribuição. Pode ser tedioso cobrir todas as variações no software dos sistemas, portanto procurei abordar as duas principais famílias de distribuição: o Debian (incluindo o Ubuntu) e o RHEL/Fedora/CentOS. O livro também está focado em instalações para desktop e servidores. Há uma quantidade significativa de informações que pode ser aplicada a sistemas embarcados como o Android e o OpenWRT, porém cabe a você descobrir as diferenças nessas plataformas.

Quais são as novidades na segunda edição?

A primeira edição deste livro (edição americana, em inglês) lidava principalmente com o aspecto centrado no usuário de um sistema Linux. O livro focava no entendimento de como as partes funcionavam e como fazê-las entrar em ação. Naquela época, muitas partes do Linux eram difíceis de ser instaladas e configuradas adequadamente.

Felizmente, esse não é mais o caso, graças ao trabalho árduo das pessoas que implementam os softwares e criam as distribuições Linux. Com isso em mente, omiti alguns materiais mais antigos e, talvez, menos relevantes (por exemplo, uma explicação detalhada sobre impressão) em favor de uma discussão mais ampla do papel do kernel

em toda a distribuição Linux. Provavelmente, você interage muito mais com o kernel do que percebe, e tomei cuidados especiais para indicar em que pontos isso acontece.

É claro que muitos dos assuntos originais deste livro sofreram mudanças ao longo dos anos, e trabalhei arduamente para analisar o material da primeira edição em busca de atualizações. De interesse particular é a maneira como o Linux inicializa e como os dispositivos são administrados. Também tive o cuidado de reorganizar o material para que esteja de acordo com os interesses e as necessidades dos leitores atuais.

Um aspecto que não mudou foi o tamanho deste livro. Quero oferecer o material necessário para que você tenha agilidade, e isso inclui explicar determinados detalhes no caminho que podem ser difíceis de dominar; contudo não quero que você precise ser um peso-pesado para entender este livro. Quando dominar os assuntos importantes presentes aqui, você não deverá ter problemas em procurar mais detalhes e entendê-los.

Também omiti algumas informações históricas que estavam na primeira edição, principalmente para manter você focado. Se estiver interessado no Linux e na forma como ele se relaciona com a história do Unix, leia o livro *The Daemon, the Gnu, and the Penguin* (Reed Media Services, 2008), de Peter H. Salus – ele faz um ótimo trabalho em explicar como o software que usamos evoluiu com o tempo.

Uma observação sobre a terminologia

Há um bom volume de discussões sobre os nomes de determinados elementos dos sistemas operacionais. Até mesmo o próprio nome “Linux” está sujeito a isso – deveria ser apenas “Linux”, ou “GNU/Linux” para indicar que o sistema operacional também contém partes do Projeto GNU? Ao longo deste livro, procurei usar os nomes mais comuns e menos estranhos possíveis.

Agradecimentos

Agradeço a todos que ajudaram na primeira edição: James Duncan, Douglas N. Arnold, Bill Fenner, Ken Hornstein, Scott Dickson, Dan Ehrlich, Felix Lee, Scott Schwartz, Gregory P. Smith, Dan Sully, Karol Jurado e Gina Steele. Para a segunda edição, gostaria de agradecer, em especial, a Jordi Gutiérrez Hermoso pela excelente revisão técnica do livro: suas sugestões e correções tiveram um valor inestimável. Agradeço também a Dominique Poulain e a Donald Karon por oferecerem alguns feedbacks excelentes com bastante antecedência, e a Hsinju Hsieh por me aturar durante o processo de revisão deste livro.

Por fim, gostaria de agradecer a Bill Pollock, meu editor de desenvolvimento, e a Laurel Chun, minha editora de produção. Serena Yang, Alison Law e a todos os demais da No Starch Press fizeram seu excelente trabalho costumeiro ao conseguir que esta nova edição fosse publicada no prazo.

CAPÍTULO 1

Quadro geral

À primeira vista, um sistema operacional moderno como o Linux é bem complicado, com uma quantidade enorme de partes que se comunicam e executam simultaneamente. Por exemplo, um servidor web pode conversar com um servidor de banco de dados que, por sua vez, pode usar uma biblioteca compartilhada utilizada por vários outros programas. Mas como tudo isso funciona?

A maneira mais eficiente de entender o funcionamento de um sistema operacional é por meio de *abstração* – um modo elegante de dizer que podemos ignorar a maior parte dos detalhes. Por exemplo, ao andar de carro, normalmente não precisamos pensar em detalhes como nos parafusos que mantêm o motor preso dentro do carro ou nas pessoas que constroem e mantêm a estrada em que o carro trafega. Se você for um passageiro em um carro, tudo o que você realmente deverá saber é o que o carro faz (transporta você para outro lugar) e alguns aspectos básicos de como usá-lo (como manusear a porta e o cinto de segurança).

Entretanto, se você estiver dirigindo um carro, será preciso saber mais. Você deverá aprender a operar os controles (por exemplo, o volante e o pedal do acelerador) e o que fazer quando algo der errado.

Por exemplo, vamos supor que a viagem de carro seja difícil. A abstração referente a “um carro que trafega por uma estrada” pode ser dividida em três partes: um carro, uma estrada e a maneira como você está dirigindo. Isso ajuda a isolar o problema: se a estrada tiver muitos desníveis, não culpe o carro nem a maneira como você está dirigindo. Em vez disso, pode ser que você queira descobrir por que a estrada se deteriorou ou, se ela for nova, por que os trabalhadores que a construíram fizeram um trabalho ruim.

Os desenvolvedores de software usam a abstração como uma ferramenta ao criar um sistema operacional e suas aplicações. Há vários termos para uma subdivisão que use abstrações em softwares de computador, incluindo *subsistema*, *módulo* e *pacote*, porém usaremos o termo *componente* neste capítulo por causa de sua simplicidade. Ao criar um componente de software, os desenvolvedores geralmente não pensam muito na estrutura interna dos outros componentes, porém se preocupam com quais outros

componentes eles podem usar e como utilizá-los.

Este capítulo oferece uma visão bem geral dos componentes que constituem um sistema Linux. Embora cada um deles tenha uma quantidade enorme de detalhes técnicos em sua constituição interna, iremos ignorar esses detalhes e nos concentraremos no que os componentes fazem em relação ao sistema como um todo.

1.1 Níveis e camadas de abstração em um sistema Linux

Usar a abstração para dividir os sistemas de computação em componentes ajuda a compreender a situação, porém não funciona se não houver organização. Os componentes são organizados em camadas ou níveis. Uma *camada* ou *nível* corresponde a uma classificação (ou agrupamento) de um componente de acordo com a posição que esse componente ocupa entre o usuário e o hardware. Os navegadores web, os jogos e itens desse tipo estão no nível mais alto; na camada mais baixa, temos a memória no hardware do computador – os 0s e 1s. O sistema operacional ocupa a maior parte das camadas intermediárias.

Um sistema Linux tem três níveis principais. A figura 1.1 mostra esses níveis e alguns dos componentes em cada nível. O *hardware* está na base. Ele inclui a memória, assim como uma ou mais CPUs (Central Processing Units, ou Unidades centrais de processamento) para realizar processamentos, além de ler e escrever na memória. Dispositivos como discos e interfaces de rede também fazem parte do hardware.

O próximo nível acima é o *kernel*, que é o núcleo do sistema operacional. O kernel é um software que reside na memória e diz à CPU o que ela deve fazer. Ele administra o hardware e atua principalmente como uma interface entre esse e qualquer programa em execução.

Os *processos* – programas em execução administrados pelo kernel – formam coletivamente o nível mais alto do sistema, chamado de *espaço de usuário* (user space). (Um termo mais específico para processo é *processo de usuário* (user process), independentemente de um usuário interagir ou não diretamente com o processo. Por exemplo, todos os servidores web são executados como processos de usuário.)

Há uma diferença muito importante entre as maneiras como o kernel e os processos de usuário executam: o kernel executa em *modo kernel* (kernel mode) e os processos de usuário executam em *modo usuário* (user mode). Um código executando em modo kernel tem acesso irrestrito ao processador e à memória principal. É um privilégio elevado, porém perigoso, que permite a um processo do kernel provocar facilmente uma falha em todo o sistema. A área que somente o kernel pode acessar chama-se

espaço do kernel (kernel space).

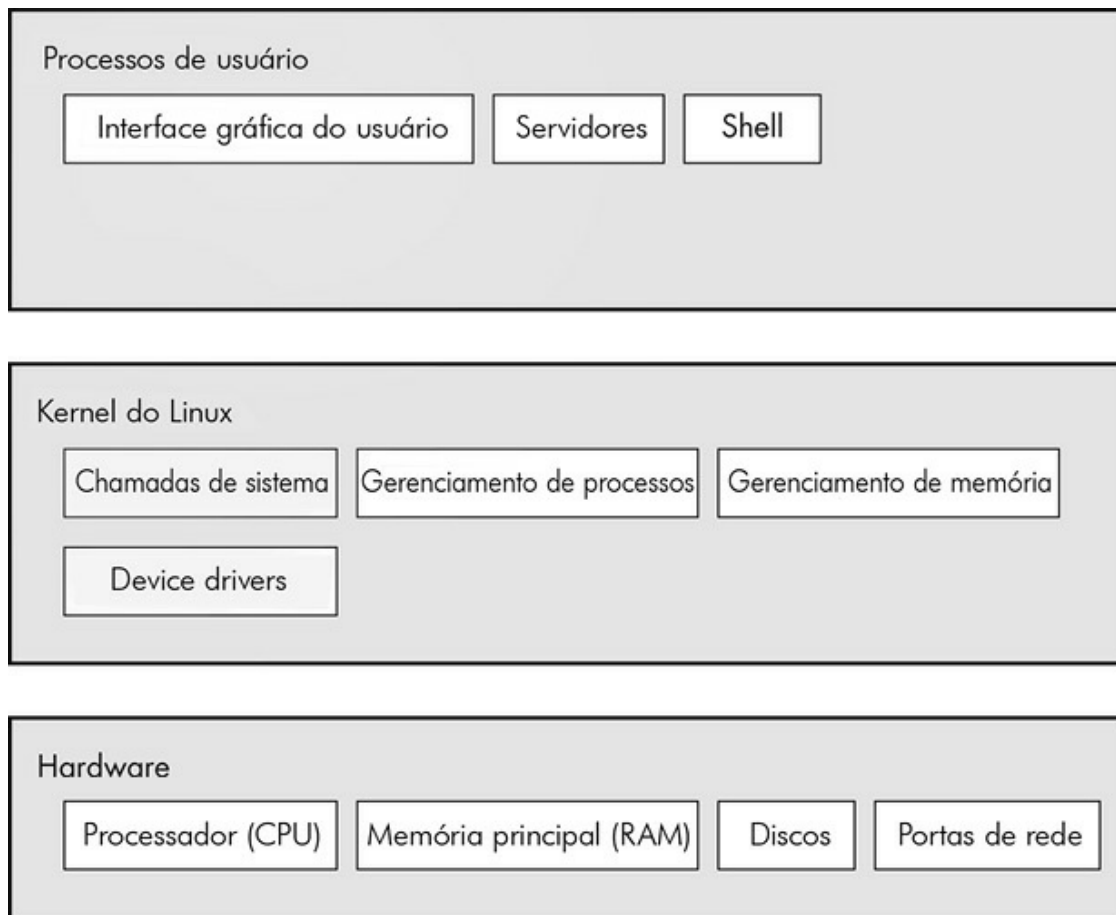


Figura 1.1 – Organização geral do sistema Linux.

Em comparação, em modo usuário, o acesso é restrito a um subconjunto (normalmente bem pequeno) de memória e de operações seguras de CPU. *O espaço de usuário* refere-se às partes da memória principal que os processos de usuário podem acessar. Se um processo cometer um erro e provocar falhas, as consequências serão limitadas e poderão ser limpas pelo kernel. Isso significa que se o seu navegador web provocar uma falha, provavelmente ele não acabará com o processamento científico que você está executando em background há dias.

Teoricamente, um processo de usuário que perca o controle não poderá causar danos sérios ao restante do sistema. Na verdade, isso depende do que você considera “danos sérios” bem como dos privilégios particulares do processo, pois alguns processos têm permissão para fazer mais do que outros. Por exemplo, um processo de usuário pode danificar completamente os dados em um disco? Com as permissões corretas, sim – e você pode considerar isso como sendo bem perigoso. Entretanto há medidas de segurança para evitar isso, e a maioria dos processos simplesmente não terá permissão para provocar falhas dessa maneira.

1.2 Hardware: compreendendo a memória principal

De todo o hardware em um sistema de computador, a *memória principal* talvez seja o mais importante. Em sua forma mais bruta, a memória principal é somente uma enorme área de armazenamento para um conjunto de 0s e 1s. Cada 0 ou 1 chama-se *bit*. É nesse local que o kernel e os processos em execução residem – eles são simplesmente coleções enormes de bits. Toda entrada e saída dos dispositivos periféricos flui pela memória principal, também na forma de um conjunto de bits. Uma CPU é somente algo que executa operações na memória: ela lê instruções e dados da memória e escreve dados de volta nela.

Com frequência, você ouvirá o termo *estado* para se referir à memória, aos processos, ao kernel e a outras partes de um sistema de computador. Estritamente falando, um estado é uma organização particular de bits. Por exemplo, se você tiver quatro bits em sua memória, 0110, 0001 e 1011 representarão três estados diferentes.

Ao considerar que um único processo pode facilmente ser constituído de milhões de bits em memória, geralmente será mais fácil usar termos abstratos para discutir esses estados. Em vez de descrever um estado usando bits, você descreverá o que algo fez ou está fazendo no momento. Por exemplo, podemos dizer que “o processo está aguardando uma entrada” ou que “o processo está executando o Estágio 2 de sua inicialização”.

👍 **Observação:** Pelo fato de ser comum referir-se ao estado em termos abstratos em vez de usar os bits propriamente ditos, o termo *imagem* refere-se a uma organização física em particular dos bits.

1.3 Kernel

Por que estamos falando de memória principal e de estados? Praticamente tudo que o kernel faz gira em torno da memória principal. Uma das tarefas do kernel consiste em separar a memória em várias subdivisões, e ele deve manter determinadas informações de estado sobre essas subdivisões o tempo todo. Cada processo obtém sua própria quota de memória, e o kernel deve garantir que cada processo atenha-se à sua quota.

O kernel é responsável pelo gerenciamento das tarefas em quatro áreas gerais do sistema:

- Processos – o kernel é responsável por determinar quais processos têm permissão para usar a CPU.
- Memória – o kernel deve ajudar a monitorar toda a memória – que parte está alocada no momento para um processo em particular, que parte deve ser

compartilhada entre os processos e que parte está livre.

- Device drivers – o kernel atua como uma interface entre o hardware (por exemplo, um disco) e os processos. Normalmente, operar o hardware é uma tarefa do kernel.
- Chamadas de sistema e suporte – os processos normalmente usam chamadas de sistema para se comunicar com o kernel.

Iremos agora explorar rapidamente cada uma dessas áreas.

👍 **Observação:** se você estiver interessado no funcionamento detalhado de um kernel, dois bons livros-texto são: *Operating System Concepts*, 9ª edição, de Abraham Silberschatz, Peter B. Galvin e Greg Gagne (Wiley, 2012), e *Modern Operating Systems*, 4ª edição, de Andrew S. Tanenbaum e Herbert Bos (Prentice Hall, 2014).

1.3.1 Gerenciamento de processos

O *gerenciamento de processos* descreve a inicialização, a pausa, a retomada e o término dos processos. Os conceitos por trás da inicialização e do término de processos são bem simples, porém descrever como um processo utiliza a CPU em seu curso normal de operação é um pouco mais complicado.

Em qualquer sistema operacional moderno, muitos processos executam “simultaneamente”. Por exemplo, você pode ter um navegador web e uma planilha abertos em um computador desktop ao mesmo tempo. Entretanto a situação não é o que aparenta ser: os processos por trás dessas aplicações normalmente não executam *exatamente* ao mesmo tempo.

Considere um sistema com uma CPU. Muitos processos *podem* usar a CPU, porém somente um processo realmente utilizará a CPU em um determinado instante. Na prática, cada processo utiliza a CPU durante uma pequena fração de segundo e depois faz uma pausa; em seguida, outro processo usa a CPU durante outra pequena fração de segundo e então é a vez de outro processo e assim por diante. O ato de um processo passar o controle da CPU para outro processo chama-se *alternância de contexto* (context switch).

Cada porção de tempo – chamada de *time slice* (fatia de tempo) – dá a um processo tempo suficiente para efetuar um processamento significativo (de fato, um processo normalmente termina sua tarefa corrente em uma única fatia de tempo). No entanto, como as fatias de tempo são bem curtas, os seres humanos não conseguem percebê-las e o sistema parecerá estar executando vários processos ao mesmo tempo [uma capacidade conhecida como *multitarefa* (multitasking)].

O kernel é responsável pela alternância de contexto. Para entender como isso funciona,

vamos pensar em uma situação em que um processo esteja executando em modo usuário, porém seu time slice tenha esgotado. Eis o que acontece:

1. A CPU (o hardware propriamente dito) interrompe o processo corrente de acordo com um temporizador interno, alterna para o modo kernel e devolve o controle ao kernel.
2. O kernel registra os estados correntes da CPU e da memória, que serão essenciais para retomar o processo que acabou de ser interrompido.
3. O kernel executa qualquer tarefa que possa ter surgido durante o time slice anterior [por exemplo, coletar dados de operações de entrada e saída (I/O)].
4. O kernel agora está pronto para deixar outro processo executar. Ele analisa a lista de processos que estão prontos para executar e seleciona um.
5. O kernel prepara a memória para esse novo processo e, em seguida, prepara a CPU.
6. O kernel informa à CPU quanto tempo irá durar o time slice do novo processo.
7. O kernel muda a CPU para o modo usuário e passa o controle da CPU para o processo.

A alternância de contexto responde à importante pergunta sobre *quando* o kernel é executado. A resposta é que ele executa *entre* os time slices dos processos, durante uma alternância de contexto.

No caso de um sistema com várias CPUs, a situação é um pouco mais complicada, pois o kernel não precisa ceder o controle de sua CPU corrente para permitir que um processo execute em uma CPU diferente. No entanto, para maximizar o uso de todas as CPUs disponíveis, o kernel normalmente faz isso de qualquer modo (e pode usar determinados truques para conseguir um pouco mais de tempo de CPU para si mesmo).

1.3.2 Gerenciamento de memória

Como o kernel deve administrar a memória durante uma alternância de contexto, ele tem uma tarefa complexa de gerenciamento de memória. O trabalho do kernel é complicado porque as seguintes condições devem ser atendidas:

- O kernel deve ter sua própria área privada na memória, à qual os processos de usuário não deverão ter acesso.
- Cada processo de usuário deve ter sua própria seção de memória.
- Um processo de usuário não poderá ter acesso à memória privada de outro

processo.

- Os processos de usuário podem compartilhar memória.
- Partes da memória dos processos de usuário podem ser somente para leitura.
- O sistema pode usar mais memória do que está fisicamente presente ao usar o espaço em disco como auxiliar.

Felizmente para o kernel, existe ajuda. As CPUs modernas incluem uma MMU (Memory Management Unit, ou Unidade de gerenciamento de memória) que permite ter um esquema de acesso à memória chamado *memória virtual*. Ao usar a memória virtual, um processo não acessa diretamente a memória por meio de sua localização física no hardware. Em vez disso, o kernel configura cada processo para que atue como se ele tivesse todo o computador para si mesmo. Quando o processo acessa alguma parte de sua memória, a MMU intercepta o acesso e utiliza um mapeamento de endereços de memória para traduzir o endereço de memória do processo para um endereço realmente físico na memória do computador. O kernel deve ainda inicializar além de continuamente manter e alterar esse mapeamento de endereços de memória. Por exemplo, durante uma alternância de contexto, o kernel deve alterar o mapeamento, do processo de saída para o processo de entrada.

 **Observação:** a implementação de um mapeamento de endereços de memória chama-se tabela de páginas (page table).

Você aprenderá mais sobre como verificar o desempenho da memória no capítulo 8.

1.3.3 Device drivers e gerenciamento de dispositivos

A função do kernel em relação aos dispositivos é bem simples. Um dispositivo normalmente é acessível somente em modo kernel, pois acessos impróprios (por exemplo, um processo de usuário que solicite um desligamento de energia) poderiam provocar falhas no computador. Outro problema é que dispositivos diferentes raramente têm a mesma interface de programação, mesmo que os dispositivos realizem a mesma tarefa, por exemplo, duas placas de rede diferentes. Sendo assim, os device drivers, tradicionalmente, têm feito parte do kernel e se esforçam em apresentar uma interface uniforme aos processos de usuário para simplificar o trabalho dos desenvolvedores de software.

1.3.4 Chamadas de sistema e suporte

Há vários outros tipos de recurso disponibilizados pelo kernel aos processos de usuário. Por exemplo, as *chamadas de sistema* (ou *syscalls*) executam tarefas

específicas que um processo de usuário sozinho não poderá efetuar ou não poderá fazê-lo adequadamente. Por exemplo, todas as ações referentes a abrir, ler e escrever em arquivos envolvem chamadas de sistema.

Duas chamadas de sistema, `fork()` e `exec()`, são importantes para entender como os processos são inicializados:

- `fork()` – quando um processo chama `fork()`, o kernel cria uma cópia praticamente idêntica do processo.
- `exec()` – quando um processo chama `exec(programa)`, o kernel inicia *programa*, substituindo o processo corrente.

Além de `init` (veja o capítulo 6), *todos* os processos de usuário em um sistema Linux começam como resultado de `fork()`, e, na maioria das vezes, você também executará `exec()` para iniciar um novo programa em vez de executar uma cópia de um processo existente. Um exemplo bem simples é qualquer programa que seja executado na linha de comando, por exemplo, o comando `ls` para mostrar o conteúdo de um diretório. Ao digitar `ls` em uma janela de terminal, o shell que estiver executando na janela do terminal chamará `fork()` para criar uma cópia do shell e, em seguida, a nova cópia do shell chamará `exec(ls)` para executar `ls`. A figura 1.2 mostra o fluxo dos processos e das chamadas de sistema para iniciar um programa como o `ls`.

👍 **Observação:** as chamadas de sistema normalmente são indicadas com parênteses. No exemplo mostrado na figura 1.2, o processo que está pedindo ao kernel que crie outro processo deve executar uma chamada de sistema `fork()`. Essa notação resulta da maneira como a chamada seria escrita na linguagem de programação C. Não é preciso conhecer C para entender este livro; basta lembrar que uma chamada de sistema consiste em uma interação entre um processo e o kernel. Além disso, este livro simplifica determinados grupos de chamadas de sistema. Por exemplo, `exec()` se refere a toda uma família de chamadas de sistema que realiza uma tarefa semelhante, porém difere quanto à programação.

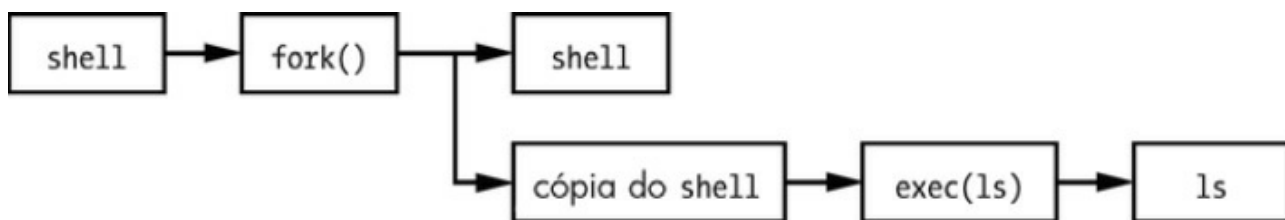


Figura 1.2 – Iniciando um novo processo.

O kernel também suporta processos de usuário com recursos diferentes das chamadas de sistema tradicionais, e o mais comum deles são os *pseudodispositivos* (pseudodevices). Os pseudodispositivos se parecem com os dispositivos para os processos de usuário, porém são puramente implementados em software. Sendo assim, tecnicamente, eles não precisariam estar no kernel, mas normalmente estão lá por razões práticas. Por exemplo, o dispositivo gerador de número aleatório do kernel

(*/dev/random*) seria difícil de ser implementado de forma segura com um processo de usuário.

👍 **Observação:** tecnicamente, um processo de usuário que acesse um pseudodispositivo deve continuar usando uma chamada de sistema para abrir o dispositivo, portanto os processos não poderão evitar totalmente as chamadas de sistema.

1.4 Espaço de usuário

Conforme mencionamos anteriormente, a memória principal que o kernel aloca para os processos de usuário chama-se *espaço de usuário* (user space). Como um processo é simplesmente um estado (ou uma imagem) em memória, o espaço de usuário também se refere à memória para todo o conjunto de processos em execução. [Você também poderá ouvir o termo mais informal *userland* (território do usuário) usado para se referir ao espaço de usuário.]

A maior parte da verdadeira ação em um sistema Linux ocorre no espaço de usuário. Embora todos os processos sejam essencialmente iguais do ponto de vista do kernel, eles executam tarefas diferentes para os usuários. Há uma estrutura rudimentar de níveis (ou camadas) de serviços para os tipos de componentes de sistema que os processos de usuário representam. A figura 1.3 mostra como um conjunto de componentes de exemplo se encaixa e interage em um sistema Linux. Os serviços básicos estão no nível inferior (mais próximos ao kernel), os serviços utilitários estão no meio e as aplicações em contato com os usuários estão no nível superior. A figura 1.3 é um diagrama bastante simplificado, pois somente seis componentes estão sendo mostrados; no entanto você pode ver que os componentes na parte superior estão mais próximos do usuário (a interface de usuário e o navegador web); os componentes do nível intermediário contêm um servidor de emails utilizado pelo navegador web e há diversos componentes menores na parte inferior.

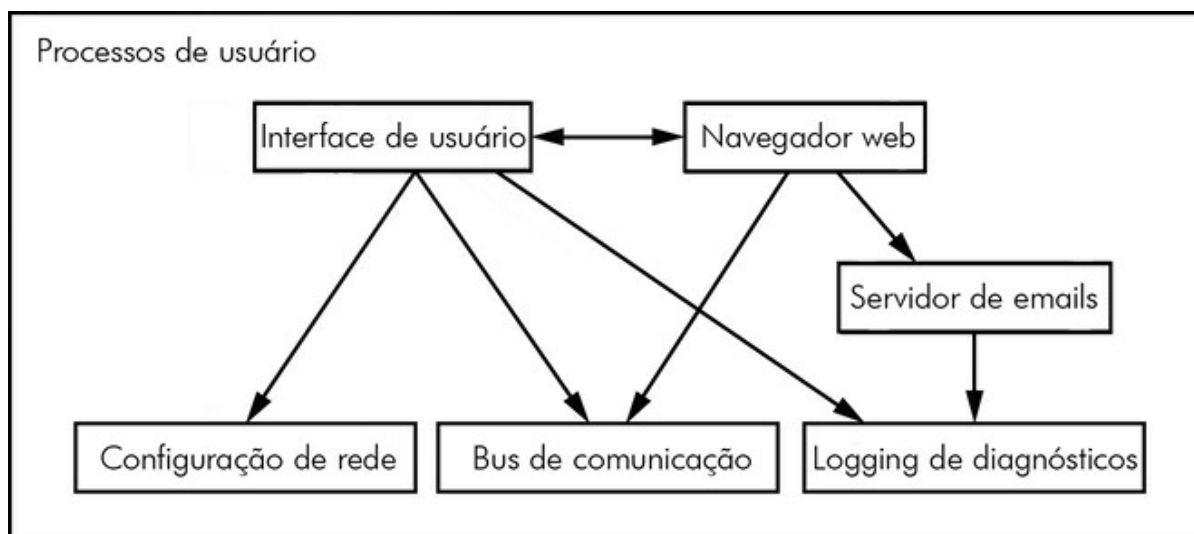


Figura 1.3 – Tipos de processo e interações.

O nível inferior tende a ser constituído de pequenos componentes que realizam tarefas únicas e descomplicadas. O nível intermediário contém componentes maiores como emails, impressão e serviços de banco de dados. Por fim, os componentes no nível superior executam tarefas complexas, normalmente controladas diretamente pelo usuário. Os componentes também usam outros componentes. Em geral, se um componente quiser usar outro, o segundo componente estará no mesmo nível de serviço ou abaixo dele.

Entretanto a figura 1.3 é somente uma aproximação de uma organização do espaço de usuário. Na realidade, não há regras nesse espaço. Por exemplo, a maioria das aplicações e dos serviços escreve mensagens de diagnóstico conhecidas como *logs*. Boa parte dos programas usa o serviço-padrão syslog para escrever mensagens de log, porém algumas pessoas preferem fazer todo o logging por conta própria.

Além do mais, é difícil classificar alguns componentes do espaço de usuário. Os componentes de servidor, como os servidores web e de banco de dados, podem ser considerados como aplicações de nível bem alto, pois suas tarefas normalmente são complexas, portanto você poderá colocá-los no nível superior na figura 1.3. Entretanto as aplicações de usuário podem depender desses servidores para realizar tarefas que elas prefeririam não realizar por conta própria, portanto seria possível argumentar também que os servidores poderiam ser colocados no nível intermediário.

1.5 Usuários

O kernel do Linux suporta o conceito tradicional de um usuário Unix. Um *usuário* é uma entidade que pode executar processos e ser dono de arquivos. Um usuário está

associado a um *nome de usuário*. Por exemplo, um sistema pode ter um usuário chamado *billyjoe*. No entanto o kernel não administra os nomes de usuário; em vez disso, ele identifica os usuários por meio de identificadores numéricos simples chamados de *userids*. (Você aprenderá mais sobre a correspondência entre nomes de usuário e os *userids* no capítulo 7.)

Os usuários existem principalmente para dar suporte a permissões e limitações. Todo processo no espaço de usuário tem um usuário que é o seu *proprietário* (owner), e dizemos que os processos executam *como* proprietário. Um usuário pode terminar ou modificar o comportamento de seus próprios processos (dentro de determinados limites), porém não poderá interferir nos processos de outros usuários. Além disso, os usuários podem ser proprietários de arquivos e optar por compartilhá-los ou não com outros usuários.

Um sistema Linux normalmente tem diversos usuários, além daqueles que correspondem a seres humanos de verdade que usem o sistema. Você os verá com mais detalhes no capítulo 3, mas o usuário mais importante a se conhecer é o *root*. O usuário *root* é uma exceção às regras anteriores, pois o *root* pode terminar e alterar os processos de outros usuários, além de ler qualquer arquivo do sistema local. Por esse motivo, o *root* é conhecido como *superusuário* (superuser). Dizemos que uma pessoa que atue como *root* tem *acesso de root* e é um administrador em um sistema Unix tradicional.

👍 **Observação:** atuar como *root* pode ser perigoso. Pode ser difícil identificar e corrigir erros, pois o sistema permitirá que você faça de tudo, mesmo que o que você fizer seja prejudicial ao sistema. Por esse motivo, os designers do sistema tentam constantemente fazer com que o acesso de *root* seja tão desnecessário quanto possível, por exemplo, não exigindo esse tipo de acesso para alternar entre redes wireless em um notebook. Além disso, por mais poderoso que seja o usuário *root*, ele continuará executando no modo usuário do sistema operacional, e não em modo kernel.

Os *grupos* correspondem a conjuntos de usuários. O principal propósito dos grupos é permitir que um usuário compartilhe acesso a arquivos com outros usuários em um grupo.

1.6 Próximos passos

Até agora, você viu o que compõe um sistema Linux *em execução*. Os processos de usuário formam o ambiente com o qual você interage diretamente; o kernel administra os processos e o hardware. Tanto o kernel quanto os processos residem na memória.

São informações básicas ótimas, porém você não poderá conhecer os detalhes de um sistema Linux somente lendo a seu respeito; será necessário pôr a mão na massa. O próximo capítulo dará início à sua jornada ao ensinar alguns conceitos básicos sobre o

espaço de usuário. Ao longo do caminho, você conhecerá uma parte importante do sistema Linux que não foi discutida neste capítulo – o armazenamento de longo prazo (discos, arquivos etc.). Afinal de contas, você deverá armazenar os seus programas e os dados em algum lugar.

CAPÍTULO 2

Comandos básicos e hierarquia de diretórios

Este capítulo é um guia para os comandos e utilitários Unix que serão referenciados ao longo deste livro. É um material preliminar e pode ser que um volume substancial dele já seja de seu conhecimento. Mesmo que você ache que já esteja preparado, reserve alguns instantes para passar os olhos pelas páginas do capítulo somente por garantia, especialmente quando se tratar do material referente à hierarquia de diretórios na seção 2.19.

Por que falar de comandos Unix? Este livro não é sobre o funcionamento do Linux? É claro que é, porém o Linux, no fundo, é uma variação do Unix. Neste capítulo, você verá a palavra *Unix* com mais frequência do que *Linux* e poderá usar o que aprender diretamente no Solaris, no BSD e em outros sistemas do tipo Unix. Procurei evitar discutir muitas extensões de interface de usuário específicas do Linux não só para proporcionar uma base melhor para usar os outros sistemas operacionais, mas também porque essas extensões tendem a ser instáveis. Você poderá adaptar-se a novas versões de Linux muito mais rapidamente se conhecer os comandos essenciais.

 **Observação:** para obter mais detalhes sobre o Unix para iniciantes, além daqueles encontrados neste livro, considere a leitura de *The Linux Command Line* (No Starch Press, 2012), *UNIX for the Impatient* (Addison-Wesley Professional, 1995) e *Learning the UNIX Operating System*, 5ª edição (O'Reilly, 2001).

2.1 Bourne shell: `/bin/sh`

O shell é uma das partes mais importantes de um sistema Unix. Um *shell* é um programa que executa comandos, como aqueles digitados pelos usuários. O shell também serve como um pequeno ambiente de programação. Os programadores Unix normalmente dividem tarefas comuns em pequenos componentes e usam o shell para administrar tarefas e reunir tudo.

Muitas partes importantes do sistema na realidade são *shell scripts* – arquivos-texto que contêm uma sequência de comandos de shell. Se você já trabalhou anteriormente com MS-DOS, poderá pensar nos shell scripts como arquivos *.BAT* bastante eficazes. Em virtude de sua importância, o capítulo 11 é dedicado totalmente aos shell scripts.

À medida que avançar neste livro e adquirir prática, você aperfeiçoará o seu

conhecimento sobre manipulação de comandos usando o shell. Uma das melhores características do shell está no fato de que, se você cometer um erro, poderá ver facilmente o que foi digitado para descobrir o que está errado e então tentar novamente.

Há vários shells Unix diferentes, porém todos derivam vários de seus recursos do Bourne shell (*/bin/sh*) – um shell-padrão desenvolvido pelo Bell Labs para as primeiras versões de Unix. Todo sistema Unix precisa do Bourne shell para funcionar corretamente, como você verá ao longo deste livro.

O Linux utiliza uma versão melhorada do Bourne shell chamada *bash* – ou “Bourne-again” shell. O shell *bash* é default na maioria das distribuições Linux, e */bin/sh* normalmente é um link para *bash* em um sistema Linux. Use o shell *bash* quando executar os exemplos deste livro.

 **Observação:** pode ser que o *bash* não seja o seu shell default se você estiver usando este capítulo como guia para trabalhar com uma conta Unix em uma empresa em que você não seja o administrador do sistema. Você poderá alterar o seu shell usando *chsh*, ou poderá pedir ajuda ao seu administrador de sistema.

2.2 Usando o shell

Ao instalar o Linux, crie pelo menos um usuário normal, além do usuário *root*; essa será sua conta pessoal. Para este capítulo, você deverá fazer login com seu usuário normal.

2.2.1 Janela do shell

Após fazer login, abra uma janela de shell (normalmente chamada de *terminal*). A maneira mais fácil de fazer isso a partir de uma GUI como o Gnome ou o Unity do Ubuntu é abrir uma aplicação de terminal, que inicia um shell dentro de uma nova janela. Após ter aberto um shell, ele deverá exibir um prompt na parte superior, que normalmente termina com um sinal de dólar (\$). No Ubuntu, esse prompt deve se parecer com *nome@host:path\$*, e no Fedora, é *[nome@host path]\$*. Se você estiver familiarizado com o Windows, a janela de shell terá uma aparência semelhante ao prompt de comandos DOS; a aplicação Terminal no OS X é essencialmente igual a uma janela de shell do Linux.

Este livro contém diversos comandos que você digitará em um prompt do shell. Todos eles começam com um único \$ para indicar esse prompt. Por exemplo, digite o comando a seguir (somente a parte em negrito, e não o \$) e tecle Enter:

```
$ echo Hello there.
```

 **Observação:** muitos comandos de shell neste livro começam com #. Você deverá executar esses comandos como superusuário (*root*). Eles normalmente exigem cuidado extra.

Agora digite este comando:

```
$ cat /etc/passwd
```

Este comando exibe o conteúdo do arquivo */etc/passwd*, que contém informações do sistema e, em seguida, o seu prompt de shell será retornado. Não se preocupe com o que esse arquivo faz no momento; você aprenderá sobre ele mais tarde, no capítulo 7.

2.2.2 cat

O comando `cat` é um dos comandos Unix mais fáceis de entender; ele simplesmente exibe o conteúdo de um ou mais arquivos. A sintaxe geral do comando `cat` é:

```
$ cat arquivo1 arquivo2 ...
```

Ao executar esse comando, `cat` exibe o conteúdo de *arquivo1*, *arquivo2* e de qualquer outro arquivo que você especificar (indicado por ...), e depois terminará. O comando chama-se `cat` porque ele faz uma concatenação ao exibir o conteúdo de mais de um arquivo.

2.2.3 Entrada e saída padrão

Usaremos `cat` para explorar brevemente a entrada e a saída (I/O) do Unix. Os processos Unix usam *streams* (fluxos) de I/O para ler e escrever dados. Os processos leem dados dos streams de entrada e escrevem dados nos streams de saída. Os streams são bem flexíveis. Por exemplo, a fonte de um stream de entrada pode ser um arquivo, um dispositivo, um terminal ou até mesmo o stream de saída de outro processo.

Para ver um stream de entrada em ação, digite `cat` (sem nomes de arquivo) e tecele Enter. Dessa vez, você não terá de volta o seu prompt de shell, pois `cat` ainda estará executando. Agora digite qualquer dado e tecele Enter no final de cada linha. O comando `cat` repetirá cada linha que você digitar. Após ter ficado suficientemente entediado, tecele Ctrl-D em uma linha vazia para finalizar o `cat` e retornar ao prompt de shell.

O motivo de o `cat` ter adotado um comportamento interativo tem a ver com os streams. Pelo fato de não ter especificado um nome de arquivo de entrada, `cat` leu do stream da *entrada-padrão* (standard input) disponibilizado pelo kernel do Linux, em vez de ler de um stream conectado a um arquivo. Nesse caso, a entrada-padrão estava conectada ao terminal em que `cat` foi executado.

👍 **Observação:** Teclar Ctrl-D em uma linha vazia interrompe a entrada-padrão corrente do terminal (e geralmente encerra um programa). Não o confunda com Ctrl-C, que encerra um programa independentemente de sua entrada ou de sua saída.

A saída-padrão (standard output) é semelhante. O kernel disponibiliza a cada

processo um stream de saída-padrão em que ele poderá escrever sua saída. O comando `cat` sempre escreve sua saída na saída-padrão. Ao executar `cat` no terminal, a saída-padrão estava conectada a esse terminal, portanto foi aí que você viu a saída.

A saída e a entrada padrão frequentemente são abreviadas como *stdin* e *stdout*. Muitos comandos funcionam como o `cat`; se um arquivo de entrada não for especificado, o comando lerá de *stdin*. A saída é um pouco diferente. Alguns comandos (como `cat`) enviam a saída somente para *stdout*, porém outros têm a opção de enviá-la diretamente para arquivos.

Há um terceiro stream-padrão de I/O chamado erro-padrão (standard error). Você o verá na seção 2.14.1.

Um dos melhores recursos dos streams-padrão é poder manipulá-los facilmente para que leiam e escrevam em locais que não sejam o terminal, como você verá na seção 2.14. Em particular, você aprenderá a conectar streams a arquivos e a outros processos.

2.3 Comandos básicos

Vamos agora dar uma olhada em mais alguns comandos Unix. A maioria dos programas a seguir aceita vários argumentos, e alguns têm tantas opções e tantos formatos que uma lista completa não faria sentido. Esta é uma lista simplificada dos comandos básicos; você ainda não precisará de todos os detalhes.

2.3.1 `ls`

O comando `ls` lista o conteúdo de um diretório. O default é o diretório corrente. Use `ls -l` para obter uma listagem detalhada (longa) e `ls -F` para exibir informações sobre o tipo de arquivo. (Para obter mais informações sobre os tipos e as permissões de arquivo exibidos na coluna à esquerda mostrada a seguir, consulte a seção 2.17.) Aqui está um exemplo de listagem longa; ela inclui o proprietário do arquivo (coluna 3), o grupo (coluna 4), o tamanho do arquivo (coluna 5) e a data/hora da modificação (entre a coluna 5 e o nome do arquivo):

```
$ ls -l
total 3616
-rw-r--r-- 1 juser  users      3804 Apr 30  2011 abusive.c
-rw-r--r-- 1 juser  users      4165 May 26  2010 battery.zip
-rw-r--r-- 1 juser  users    131219 Oct 26  2012 beav_1.40-13.tar.gz
-rw-r--r-- 1 juser  users      6255 May 30  2010 country.c
drwxr-xr-x 2 juser  users       4096 Jul 17 20:00 cs335
-rwxr-xr-x 1 juser  users       7108 Feb  2  2011 dhry
```

```
-rw-r--r-- 1 juser  users    11309 Oct 20 2010 dhry.c
-rw-r--r-- 1 juser  users      56 Oct  6 2012 doit
drwxr-xr-x 6 juser  users    4096 Feb 20 13:51 dw
drwxr-xr-x 3 juser  users    4096 May  2 2011 hough-stuff
```

Você aprenderá mais sobre o `d` na coluna 1 dessa saída na seção 2.17.

2.3.2 cp

Em sua forma mais simples, `cp` copia arquivos. Por exemplo, para copiar *arquivo1* para *arquivo2*, digite:

```
$ cp arquivo1 arquivo2
```

Para copiar vários arquivos para um diretório (pasta) chamado *dir*, experimente usar este comando:

```
$ cp arquivo1 ... arquivoN dir
```

2.3.3 mv

O comando `mv` (mover) é como `cp`. Em sua forma mais simples, o comando renomeia um arquivo. Por exemplo, para renomear *arquivo1* para *arquivo2*, digite:

```
$ mv arquivo1 arquivo2
```

`mv` também pode ser usado para mover vários arquivos para um diretório diferente:

```
$ mv arquivo1 ... arquivoN dir
```

2.3.4 touch

O comando `touch` cria um arquivo. Se o arquivo já existir, `touch` não irá alterá-lo, porém o timestamp de modificação do arquivo, exibido com o comando `ls -l`, será atualizado. Por exemplo, para criar um arquivo vazio, digite:

```
$ touch arquivo
```

Em seguida, execute `ls -l` nesse arquivo. Você deverá ver uma saída como a que se segue, em que a data e a hora ❶ indicam quando o comando `touch` foi executado.

```
$ ls -l arquivo
-rw-r--r-- 1 juser users 0 May 21 18:32❶ arquivo
```

2.3.5 rm

Para apagar (remover) um arquivo, use `rm`. Após tê-lo removido, ele será eliminado de seu sistema e, em geral, não poderá ser recuperado.

```
$ rm arquivo
```

2.3.6 echo

O comando `echo` exibe seus argumentos na saída-padrão:

```
$ echo Hello again.
```

```
Hello again.
```

O comando `echo` é muito útil para revelar expansões de globs¹ de shell (“caracteres-curinga” como `*`) e variáveis (como `$HOME`), que você verá mais adiante neste capítulo.

2.4 Navegando pelos diretórios

O Unix tem uma hierarquia de diretórios que começa com `/` que, às vezes, é chamado de *diretório-raiz*. O separador de diretórios é a barra (`/`), e *não* a barra invertida (`\`). Há vários subdiretórios-padrão no diretório-raiz, por exemplo, `/usr`, como você verá na seção 2.19.

Ao se referir a um arquivo ou a um diretório, especifique um *path* (caminho) ou um *pathname* (nome do caminho). Quando um path começa com `/` (por exemplo, `/usr/lib`), temos um path *completo* ou *absoluto*.

Um componente do path identificado por dois pontos (`..`) especifica o pai de um diretório. Por exemplo, se você estiver trabalhando em `/usr/lib`, o path `..` refere-se a `/usr`. De modo semelhante, `../bin` se refere a `/usr/bin`.

Um ponto (`.`) refere-se ao diretório corrente; por exemplo, se você estiver em `/usr/lib`, o path ponto (`.`) continua sendo `/usr/lib`, e `./X11` é `/usr/lib/X11`. Não é preciso usar ponto (`.`) com muita frequência porque a maioria dos comandos usa o diretório corrente como default se um path não começar com `/` (você poderia simplesmente usar `X11` em vez de `./X11` no exemplo anterior).

Um path que não comece com `/` é chamado de *path relativo*. Na maioria das vezes, você trabalhará com paths relativos, pois já estará no diretório em que deverá estar ou em algum lugar próximo.

Agora que você já tem uma ideia do funcionamento básico dos diretórios, apresentaremos a seguir alguns comandos essenciais de diretório.

2.4.1 cd

O *diretório de trabalho corrente* é o diretório em que um processo (por exemplo, o shell) está no momento. O comando `cd` muda o diretório de trabalho corrente do shell:

```
$ cd dir
```

Se *dir* for omitido, o shell retornará para o seu *diretório home*, que é o diretório em que

você começou quando fez login.

2.4.2 mkdir

O comando `mkdir` cria um novo diretório *dir*:

```
$ mkdir dir
```

2.4.3 rmdir

O comando `rmdir` remove o diretório *dir*:

```
$ rmdir dir
```

Se *dir* não estiver vazio, esse comando irá falhar. Entretanto, se você for impaciente, é provável que não vá querer ter o trabalho de apagar todos os arquivos e subdiretórios dentro de *dir* previamente. `rm -rf dir` poderá ser usado para apagar um diretório e o seu conteúdo, mas tome cuidado! Esse é um dos poucos comandos que pode causar danos sérios, especialmente se for executado como superusuário. A opção `-r` especifica uma *remoção recursiva* para apagar repetidamente tudo o que estiver em *dir*, e `-f` força a operação de remoção. Não use as flags `-rf` com globs, por exemplo, com um asterisco (*). E, acima de tudo, sempre confira o seu comando duas vezes antes de executá-lo.

2.4.4 Globbing no shell (caracteres-curinga)

O shell pode efetuar correspondência de padrões simples com nomes de arquivos e de diretórios – processo conhecido como *globbing*, semelhante ao conceito de caracteres-curinga em outros sistemas. O mais simples dos caracteres de glob é o *, que diz ao shell para efetuar a correspondência com qualquer quantidade de caracteres arbitrários. Por exemplo, o comando a seguir exhibe os arquivos do diretório corrente:

```
$ echo *
```

O shell faz a correspondência dos argumentos contendo globs com nomes de arquivo, substitui os argumentos por esses nomes e, em seguida, executa a linha de comando revisada. A substituição é chamada de *expansão* porque o shell faz a substituição por todos os nomes de arquivo correspondentes. Eis algumas maneiras de usar * para expandir nomes de arquivo:

- `at*` é expandido com todos os nomes de arquivo que comecem com `at`.
- `*at` é expandido com todos os nomes de arquivo que terminem com `at`.
- `*at*` é expandido com todos os nomes de arquivo que contenham `at`.

Se nenhum arquivo corresponder a um glob, o shell não realizará nenhuma expansão, e

o comando será executado com os caracteres literais, como *. Por exemplo, tente executar um comando do tipo `echo *dfkdsafh`.

 **Observação:** se estiver acostumado com o MS-DOS, você poderá digitar *.* instintivamente para efetuar a correspondência com todos os arquivos. Acabe agora com esse hábito. No Linux e em outras versões do Unix, use * para efetuar a correspondência com todos os arquivos. No shell Unix, *.* corresponderá somente aos arquivos e diretórios que contenham o caractere ponto (.) em seus nomes. Nomes de arquivo no Unix não precisam ter extensões e, com frequência, não as incluem.

Outro caractere de glob de shell – o ponto de interrogação (?) – instrui o shell a efetuar a correspondência exata com um caractere arbitrário. Por exemplo, `b?at` corresponde a `boat` e a `brat`.

Se não quiser que o shell faça a expansão de um glob em um comando, coloque o glob entre aspas simples ('). Por exemplo, o comando `echo '*'` exibe um asterisco. Você achará isso prático para alguns comandos descritos na próxima seção, por exemplo, `grep` e `find`. (Você aprenderá mais sobre o uso de aspas na seção 11.2)

 **Observação:** é importante lembrar-se de que o shell realiza as expansões *antes* de executar os comandos, e *somente* nessa ocasião. Desse modo, se um * chegar até um comando sem que tenha sido expandido, o shell não fará mais nada com ele; cabe ao comando decidir o que ele deseja fazer.

Há mais a dizer sobre os recursos oferecidos por um shell moderno para efetuar correspondência de padrões, porém * e ? são aqueles que você deve conhecer no momento.

2.5 Comandos intermediários

As seções a seguir descrevem os comandos Unix intermediários essenciais.

2.5.1 grep

O comando `grep` exibe as linhas de um arquivo ou de um stream de entrada que correspondam a uma expressão. Por exemplo, para exibir as linhas do arquivo `/etc/passwd` que contenham o texto `root`, digite:

```
$ grep root /etc/passwd
```

O comando `grep` é incrivelmente prático para trabalhar com vários arquivos de uma só vez, pois ele exibe o nome do arquivo além da linha correspondente. Por exemplo, se quiser verificar todos os arquivos em `/etc` que contenham a palavra `root`, este comando poderá ser usado:

```
$ grep root /etc/*
```

Duas das opções mais importantes de `grep` são `-i` (para correspondências que não levem

em conta a diferença entre letras maiúsculas e minúsculas) e -v (que inverte a pesquisa, ou seja, exibe todas as linhas em que *não* há correspondência). Há também uma variante mais eficaz chamada `egrep` (que é apenas um sinônimo para `grep -E`).

O `grep` entende padrões conhecidos como *expressões regulares*, enraizados na teoria da ciência da computação, e são muito comuns nos utilitários Unix. As expressões regulares são mais eficazes que os padrões com caracteres-curinga e têm uma sintaxe diferente. Há dois aspectos importantes que devem ser lembrados em relação às expressões regulares:

- `.`* corresponde a qualquer quantidade de caracteres (como `*` nos caracteres-curinga).
- `.` corresponde a um caractere qualquer.

 **Observação:** A página de manual `grep(1)` contém uma descrição detalhada das expressões regulares, porém pode ser um pouco difícil lê-la. Para aprender mais, você pode consultar o livro *Dominando expressões regulares*, 3ª edição (Alta Books, 2009), ou pode consultar o capítulo sobre expressões regulares do livro *Programming Perl*, 4ª edição (O'Reilly, 2012). Se você gosta de matemática e estiver interessado na origem das expressões regulares, dê uma olhada no livro *Introduction to Automata Theory, Languages, and Computation*, 3ª edição (Prentice Hall, 2006).

2.5.2 `less`

O comando `less` é prático quando um arquivo é realmente extenso ou a saída de um comando é longa demais e ultrapassa o tamanho de uma tela.

Para efetuar paginação em um arquivo extenso como `/usr/share/dict/words`, use o comando `less /usr/share/dict/words`. Ao executar `less`, você verá o conteúdo do arquivo, uma tela cheia de cada vez. Pressione a barra de espaço para avançar no arquivo e a tecla `b` para retornar uma tela cheia. Para sair, digite `q`.

 **Observação:** o comando `less` é uma versão melhorada de um programa mais antigo chamado `more`. A maioria dos desktops e servidores Linux disponibiliza `less`, porém esse comando não é padrão em muitos sistemas embarcados e em outros sistemas Unix. Portanto, se você se vir em uma situação em que `less` não possa ser usado, tente utilizar `more`.

Também é possível procurar um texto em `less`. Por exemplo, para pesquisar uma palavra que esteja adiante, digite `/word`; para pesquisar uma anterior, use `?word`. Ao encontrar uma correspondência, tecla `n` para continuar a pesquisa.

Como você verá na seção 2.14, a saída-padrão de praticamente todo programa poderá ser enviada diretamente para a entrada-padrão de outro programa. Isso é excepcionalmente útil quando houver um comando com uma saída extensa a ser analisada e você quiser usar algo como `less` para visualizar a saída. Aqui está um exemplo do envio da saída de um comando `grep` para `less`:

```
$ grep ie /usr/share/dict/words | less
```

Tente executar esse comando. É provável que você vá usar `less` dessa maneira em diversas ocasiões.

2.5.3 pwd

O programa `pwd` (print working directory, ou exibir diretório de trabalho) simplesmente mostra o nome do diretório de trabalho corrente. Você deve estar se perguntando por que isso é necessário quando a maioria das distribuições Linux configura as contas com o diretório de trabalho corrente no prompt. Há dois motivos.

Em primeiro lugar, nem todos os prompts incluem o diretório de trabalho corrente, e você pode até mesmo querer se livrar dele em seu próprio prompt, pois ele ocupa bastante espaço. Se fizer isso, `pwd` será necessário.

Em segundo lugar, os links simbólicos, sobre os quais você aprenderá na seção 2.17.2, às vezes, podem esconder o path completo verdadeiro do diretório de trabalho corrente. Use `pwd -P` para acabar com essa confusão.

2.5.4 diff

Para ver as diferenças entre dois arquivos texto, use `diff`:

```
$ diff arquivo1 arquivo2
```

Várias opções podem controlar o formato da saída, e o formato de saída default geralmente é o mais legível para os seres humanos. Entretanto a maioria dos programadores prefere a saída de `diff -u` quando precisam enviá-la para alguém, pois as ferramentas automatizadas podem fazer melhor uso dela.

2.5.5 file

Se você vir um arquivo e não estiver certo sobre o seu formato, procure usar o comando `file` para ver se o sistema consegue adivinhar:

```
$ file arquivo
```

Você poderá ficar surpreso com o que esse comando de aparência inocente pode fazer.

2.5.6 find e locate

É frustrante saber que um determinado arquivo está em uma árvore de diretório em algum lugar, porém você simplesmente não sabe onde. Execute `find` para descobrir em que local está *arquivo* em *dir*:

```
$ find dir -name arquivo -print
```

Assim como a maioria dos programas desta seção, `find` é capaz de realizar algumas operações sofisticadas. Entretanto não tente usar opções como `-exec` antes de conhecer profundamente o formato usado aqui e descobrir por que as opções `-name` e `-print` são necessárias. O comando `find` aceita caracteres especiais para correspondência de padrões, como `*`, porém você deve colocá-los entre aspas simples (`'*`') para proteger os caracteres especiais do recurso de globbing do próprio shell. (Conforme a seção 2.4.4, lembre-se de que o shell expande os globs *antes* de executar os comandos.)

A maioria dos sistemas também tem um comando `locate` para encontrar arquivos. Em vez de procurar um arquivo em tempo real, `locate` pesquisa um índice que o sistema constrói periodicamente. A pesquisa com `locate` é muito mais rápida do que com `find`, porém, se o arquivo que você estiver procurando for mais novo que o índice, `locate` não poderá encontrá-lo.

2.5.7 head e tail

Para visualizar rapidamente uma parte de um arquivo ou de um stream de dados, utilize os comandos `head` e `tail`. Por exemplo, `head /etc/passwd` mostra as dez primeiras linhas do arquivo de senhas, e `tail /etc/passwd` mostra as dez últimas linhas.

Para mudar a quantidade de linhas exibidas, utilize a opção `-n`, em que *n* é o número de linhas que você deseja ver (por exemplo, `head -5 /etc/passwd`). Para exibir as linhas começando pela linha *n*, utilize `tail +n`.

2.5.8 sort

O comando `sort` coloca rapidamente as linhas de um arquivo-texto em ordem alfanumérica. Se as linhas do arquivo começarem com números e você quiser ordená-las numericamente, utilize a opção `-n`. A opção `-r` inverte a ordenação.

2.6 Alterando sua senha e o shell

Use o comando `passwd` para modificar sua senha. Você será solicitado a fornecer a sua senha antiga e a nova senha duas vezes. Escolha uma senha que não inclua palavras de verdade em qualquer idioma, e não tente combinar palavras.

Uma das maneiras mais fáceis de criar uma boa senha é escolher uma frase, gerar um acrônimo a partir dela e, em seguida, modificar esse acrônimo com um número ou algum sinal de pontuação. Tudo o que você precisará fazer então é se lembrar da frase.

Você pode alterar o seu shell usando o comando `chsh` (para uma alternativa como `ksh` ou `tcsh`), mas tenha em mente que este livro supõe que você está executando o `bash`.

2.7 Arquivos ponto

Vá para o seu diretório home, dê uma olhada nele com `ls` e, em seguida, execute `ls -a`. Você percebeu a diferença no resultado? Ao executar `ls` sem `-a`, você não verá os arquivos de configuração chamados *arquivos ponto* (dot files). São arquivos e diretórios cujos nomes começam com ponto (.). Arquivos ponto comuns são `.bashrc` e `.login`, e há diretórios ponto também, como `.ssh`.

Não há nada de especial nos arquivos ou diretórios ponto. Alguns programas não os mostram por padrão para que você não veja uma enorme confusão quando o conteúdo de seu diretório home for exibido. Por exemplo, `ls` não listará os arquivos ponto, a menos que a opção `-a` seja usada. Além do mais, os globs de shell não farão a correspondência com arquivos ponto, a menos que você utilize explicitamente um padrão como `.*`.

 **Observação:** você poderá ter problemas com globs porque `.*` corresponde a `.` e a `..` (os diretórios corrente e pai). Pode ser que você queira usar um padrão como `.[^.]*` ou `.*?*` para obter todos os arquivos ponto, *exceto* os diretórios corrente e pai.

2.8 Variáveis de ambiente e de shell

O shell pode armazenar variáveis temporárias, chamadas *variáveis de shell*, cujos valores são strings de texto. As variáveis de shell são muito úteis para armazenar valores em scripts, e algumas variáveis de shell controlam a maneira como o shell se comporta. (Por exemplo, o shell `bash` lê a variável `PS1` antes de exibir o prompt.)

Para atribuir um valor a uma variável de shell, use o sinal de igualdade (`=`). Aqui está um exemplo simples:

```
$ STUFF=blah
```

O exemplo anterior define o valor da variável chamada `STUFF` para `blah`. Para acessar essa variável, use `$STUFF` (por exemplo, tente executar `echo $STUFF`). Você aprenderá mais sobre os diversos usos de variáveis de shell no capítulo 11.

Uma *variável de ambiente* é como uma variável de shell, porém não é específica do shell. Todos os processos de sistemas Unix armazenam variáveis de ambiente. A principal diferença entre variáveis de ambiente e de shell está no fato de o sistema operacional passar todas as variáveis de ambiente de seu shell para os programas executados pelo shell, enquanto as variáveis de shell não podem ser acessadas pelos comandos que você executar.

Atribua um valor a uma variável de ambiente por meio do comando `export` do shell. Por

exemplo, se quiser transformar a variável de shell `$STUFF` em uma variável de ambiente, utilize o seguinte:

```
$ STUFF=blah
$ export STUFF
```

As variáveis de ambiente são úteis porque muitos programas as leem para obter configurações e opções. Por exemplo, você pode colocar suas opções favoritas de linha de comando para `less` na variável de ambiente `LESS`, e `less` usará essas opções quando for executado. (Muitas páginas de manual contêm uma seção marcada como `ENVIRONMENT` que descreve essas variáveis.)

2.9 O path do comando

`PATH` é uma variável de ambiente especial que contém o *path do comando* (ou o *path*, para simplificar). Um path de comando é uma lista de diretórios do sistema em que o shell fará buscas ao tentar localizar um comando. Por exemplo, ao executar `ls`, o shell pesquisará os diretórios listados em `PATH` em busca do programa `ls`. Se houver programas com o mesmo nome em vários diretórios do path, o shell executará o primeiro programa com o qual uma correspondência for efetuada.

Se o comando `echo $PATH` for executado, você verá que os componentes do path estão separados por dois-pontos (:). Por exemplo:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin
```

Para dizer ao shell para procurar os programas em mais lugares, altere a variável de ambiente `PATH`. Por exemplo, ao usar o comando a seguir, você poderá adicionar um diretório *dir* ao início do path, de modo que o shell procurará em *dir* antes de procurar em qualquer outro diretório de `PATH`.

```
$ PATH=dir:$PATH
```

Ou você pode concatenar um nome de diretório no final da variável `PATH`, fazendo com que o shell procure em *dir* por último:

```
$ PATH=$PATH:dir
```

 **Observação:** tome cuidado ao modificar o path, pois você poderá limpar acidentalmente todo o seu path se digitar `$PATH` incorretamente. Se isso acontecer, não entre em pânico! O dano não será permanente; basta reiniciar um novo shell. (Para um efeito mais duradouro, você deverá digitá-lo incorretamente ao editar um determinado arquivo de configuração e, mesmo assim, não será difícil fazer a correção.) Uma das maneiras mais fáceis de voltar ao normal é fechar a janela de terminal corrente e iniciar outra.

2.10 Caracteres especiais

Ao discutir o Linux com outras pessoas, você deverá conhecer os nomes de alguns caracteres especiais com os quais irá se deparar. Se você se interessar por esse tipo de assunto, consulte o “Jargon File” (Arquivo de jargões, em <http://www.catb.org/jargon/html/>) ou o seu companheiro impresso, *The New Hacker's Dictionary* (MIT Press, 1996).

A tabela 2.1 descreve um conjunto selecionado de caracteres especiais, muitos dos quais você já viu neste capítulo. Alguns utilitários, por exemplo, a linguagem de programação Perl, usam quase todos esses caracteres especiais!

Tabela 2.1 – Caracteres especiais

Caractere	Nome(s)	Usos
*	asterisco	Expressões regulares, caractere de glob
.	ponto	Diretório corrente, delimitador de arquivo/nome de host
!	ponto de exclamação	Negação, histórico de comandos
	pipe	Pipes de comando
/	barra (para frente)	Delimitador de diretório, comando de pesquisa
\	barra invertida	Literais, macros (<i>jamaIS</i> para diretórios)
\$	dólar	Indicação de variável, fim de linha
'	aspa simples	Strings literais
`	crase	Substituição de comando
“	aspas duplas	Strings semiliterais
^	circunflexo	Negação, início de linha
~	til	Negação, atalho para diretório
#	sustenido, hash	Comentários, pré-processador, substituições
[]	colchetes	Intervalos
{ }	chaves	Blocos de instruções, intervalos
_	underscore, underline	Substituto para um espaço

 **Observação:** com frequência, você verá caracteres de controle marcados com um acento circunflexo;

por exemplo, ^C para Ctrl-C.

2.11 Edição de linha de comando

À medida que trabalhar no shell, observe que você pode editar a linha de comando usando as teclas de direção para a esquerda e para a direita, além de efetuar paginação pelos comandos anteriores usando as teclas para cima e para baixo. Isso é padrão na maioria dos sistemas Linux.

No entanto é uma boa ideia esquecer-se das teclas de direção e usar as sequências de teclas Ctrl no lugar. Se você aprender as sequências listadas na tabela 2.2, descobrirá que será mais fácil inserir texto nos vários programas Unix que usem essas combinações-padrões de teclas.

Tabela 2.2 – Teclas usadas na linha de comando

Teclas	Ação
Ctrl-B	Mover o cursor para a esquerda
Ctrl-F	Mover o cursor para a direita
Ctrl-P	Ver o comando anterior (ou mover o cursor para cima)
Ctrl-N	Ver o próximo comando (ou mover o cursor para baixo)
Ctrl-A	Mover o cursor para o início da linha
Ctrl-E	Mover o cursor para o fim da linha
Ctrl-W	Apagar a palavra anterior
Ctrl-U	Apagar do cursor até o início da linha
Ctrl-K	Apagar do cursor até o final da linha
Ctrl-Y	Colar o texto apagado (por exemplo, com Ctrl-U)

2.12 Editores de texto

Falando em editar, é hora de conhecer um editor. Para trabalhar seriamente com o Unix, você deve ser capaz de editar arquivos sem danificá-los. A maior parte do sistema usa arquivos de configuração em formato texto simples (como aqueles em */etc*). Não é difícil editar arquivos, porém você fará isso com tanta frequência que será preciso ter uma ferramenta eficaz para realizar esse trabalho.

Tente conhecer um dos dois editores de texto Unix que são padrão de mercado: o vi ou o Emacs. A maioria dos magos do Unix tem uma atitude religiosa em relação à escolha do editor, mas não dê ouvidos a eles. Basta escolher um para você. Se selecionar um que esteja de acordo com o modo como você trabalha, será mais fácil aprendê-lo. Basicamente, a escolha se reduz a isto:

- Se quiser ter um editor que possa fazer praticamente tudo e que tenha uma extensa ajuda online, e você não se importar com algumas digitações extras para obter esses recursos, tente o Emacs.
- Se velocidade for tudo, experimente o vi; ele funciona de forma um pouco parecida com um videogame.

O livro *Learning the vi and Vim Editors: Unix Text Processing*, 7ª edição (O'Reilly, 2008), pode dizer tudo o que você precisa saber sobre o vi. Para o Emacs, utilize o tutorial online: inicie o Emacs, tecle Ctrl-H e, em seguida, digite T. Ou leia o *GNU Emacs Manual* (Free Software Foundation, 2011).

Você poderá se sentir tentado a experimentar um editor mais amigável no início, por exemplo, o Pico ou um entre os diversos editores GUI existentes por aí; contudo, se você tiver a tendência de permanecer com a primeira ferramenta que utilizar, não vá por esse caminho.

 **Observação:** editar texto é a primeira tarefa em que você começará a perceber a diferença entre o terminal e a GUI. Os editores como o vi executam na janela do terminal e usam a interface-padrão de I/O do terminal. Os editores GUI iniciam sua própria janela e apresentam a sua própria interface, independentemente dos terminais. O Emacs executa em uma GUI por padrão, mas executará também em uma janela de terminal.

2.13 Obtendo ajuda online

Os sistemas Linux vêm com uma documentação rica. Para comandos básicos, as *páginas de manual* (ou *man pages*) informarão o que você precisa saber. Por exemplo, para ver a página de manual do comando ls, execute man da seguinte maneira:

```
$ man ls
```

A maioria das páginas de manual se concentra principalmente em informações de referência, talvez com alguns exemplos e referências cruzadas, mas nada além disso. Não espere que haja um tutorial nem um estilo literário cativante.

Quando os programas têm muitas opções, a página de manual geralmente as lista de alguma maneira sistemática (por exemplo, em ordem alfabética), porém ela não informará quais são as opções importantes. Se você for paciente, normalmente será possível encontrar o que você precisa na man page. Se for impaciente, pergunte a um

amigo – ou pague a alguém para ser seu amigo de modo que você possa lhe perguntar. Para buscar uma página do manual contendo uma palavra-chave, use a opção -k:

```
$ man -k palavra-chave
```

Isso é útil se você não souber exatamente o nome do comando que você quer. Por exemplo, se estiver procurando um comando para ordenar (sort) algo, execute:

```
$ man -k sort
--trecho omitido--
comm (1)          - compare two sorted files line by line
qsort (3)         - sorts an array
sort (1)          - sort lines of text files
sortm (1)         - sort messages
tsort (1)         - perform topological sort
-- trecho omitido--
```

A saída inclui o nome da página do manual, a seção do manual (veja a seguir) e uma breve descrição do que a página do manual contém.

 **Observação:** se tiver alguma dúvida sobre os comandos descritos nas seções anteriores, as respostas poderão ser encontradas usando o comando man.

As páginas de manual são referenciadas por meio de seções numeradas. Quando alguém se referir a uma página de manual, o número da seção aparecerá entre parênteses ao lado do nome, por exemplo, ping(8). A tabela 2.3 lista as seções e seus números.

Tabela 2.3 – Seções do manual online

Seção	Descrição
1	Comandos de usuário
2	Chamadas de sistema
3	Documentação geral da biblioteca de programação Unix
4	Interface de dispositivos e informações sobre drivers
5	Descrições de arquivo (arquivos de configuração do sistema)
6	Jogos
7	Formatos de arquivo, convenções e codificações (ASCII, sufixos e assim por diante)
8	Comandos de sistema e servidores

As seções 1, 5, 7 e 8 devem ser bons complementos para este livro. A seção 4 pode ter uso marginal, e a seção 6 seria ótima se fosse um pouco mais extensa. Provavelmente, a

seção 3 não poderá ser usada se você não for um programador, porém será possível entender parte do material da seção 2 depois que você tiver lido mais sobre chamadas de sistema neste livro.

Você pode selecionar uma página de manual pela seção, o que, às vezes, é importante, pois `man` exibe a primeira página do manual que for encontrada quando uma correspondência for feita com um determinado termo de pesquisa. Por exemplo, para ler a descrição do arquivo `/etc/passwd` (em oposição à descrição do comando `passwd`), você poderá inserir o número da seção antes do nome da página:

```
$ man 5 passwd
```

As páginas do manual abordam o essencial, porém há várias outras maneiras de obter ajuda online. Se você estiver simplesmente procurando uma determinada opção de um comando, tente especificar o nome do comando, seguido de `--help` ou `-h` (a opção varia de comando para comando). Você poderá obter uma quantidade enorme de informações (como no caso de `ls --help`), ou poderá encontrar exatamente o que estava procurando.

Há algum tempo, o Projeto GNU decidiu que não gostava muito das páginas de manual e mudou para outro formato chamado *info* (ou *texinfo*). Com frequência, essa documentação vai além do que contém uma página de manual típica, mas, às vezes, é mais complexa.

Para acessar um manual *info*, utilize `info` com o nome do comando:

```
$ info comando
```

Alguns pacotes descarregam a documentação disponível em `/usr/share/doc`, sem se importar com sistemas de manual online como `man` ou `info`. Verifique esse diretório em seu sistema se você se vir procurando documentações. E, é claro, pesquise na Internet.

2.14 Entrada e saída de shell

Agora que você está familiarizado com os comandos, arquivos e diretórios básicos do Unix, estará pronto para aprender a redirecionar a entrada e a saída padrão. Vamos começar com a saída-padrão.

Para enviar a saída de *comando* para um arquivo em vez de enviar para o terminal, use o caractere de redirecionamento `>`:

```
$ comando > arquivo
```

O shell cria *arquivo*, caso ele ainda não exista. Se *arquivo* existir, o shell apagará o arquivo original antes (*sobrescreverá*). (Alguns shells têm parâmetros que evitam a sobrescrita. Por exemplo, digite `set -C` para evitar a sobrescrita no `bash`.)

A saída pode ser concatenada ao arquivo em vez de sobrescrevê-lo, usando a sintaxe de redirecionamento >>:

```
$ comando >> arquivo
```

Essa é uma maneira prática de reunir a saída em um só local ao executar sequências de comandos relacionados.

Para enviar a saída-padrão de um comando para a entrada-padrão de outro, utilize o caractere de pipe (|). Para ver como isso funciona, experimente executar estes dois comandos:

```
$ head /proc/cpuinfo
```

```
$ head /proc/cpuinfo | tr a-z A-Z
```

A saída pode ser enviada para qualquer quantidade desejada de comandos com pipe; basta adicionar outro pipe antes de cada comando adicional.

2.14.1 Erro-padrão

Ocasionalmente, você poderá redirecionar a saída-padrão, porém perceberá que o programa continua exibindo informações no terminal. Isso é chamado de *erro-padrão* (stderr); é um stream de saída adicional para diagnósticos e depuração. Por exemplo, este comando gera um erro:

```
$ ls /ffffff > f
```

Após o comando ter sido concluído, *f* deverá estar vazio, porém você continuará vendo a mensagem de erro a seguir no terminal como erro-padrão:

```
ls: cannot access /ffffff: No such file or directory
```

Se quiser, você poderá redirecionar o erro-padrão. Por exemplo, para enviar a saída-padrão para *f* e o erro-padrão para *e*, use a sintaxe >>, desta maneira:

```
$ ls /ffffff > f 2> e
```

O número 2 especifica o *ID do stream* modificado pelo shell. O ID de stream 1 corresponde à saída-padrão (default), e 2 é o erro-padrão.

Você também pode enviar o erro-padrão para o mesmo local que stdout usando a notação >&. Por exemplo, para enviar tanto a saída-padrão quanto o erro-padrão para o arquivo chamado *f*, experimente executar este comando:

```
$ ls /ffffff > f 2>&1
```

2.14.2 Redirecionamento da entrada-padrão

Para canalizar um arquivo para a entrada-padrão de um programa, utilize o operador <:


```
$ head < /proc/cpuinfo
```

Ocasionalmente, você irá se deparar com um programa que exija esse tipo de redirecionamento, porém, pelo fato de a maioria dos comandos Unix aceitar nomes de arquivo como argumento, isso não será muito comum. Por exemplo, o comando anterior poderia ter sido escrito como `head /proc/cpuinfo`.

2.15 Entendendo as mensagens de erro

Ao encontrar um problema em um sistema do tipo Unix, por exemplo, o Linux, você *deve* ler a mensagem de erro. De modo diferente das mensagens de outros sistemas operacionais, os erros do Unix normalmente informam exatamente o que deu errado.

2.15.1 Anatomia de uma mensagem de erro UNIX

A maioria dos programas Unix gera e informa as mesmas mensagens de erro básicas, porém pode haver diferenças sutis entre a saída de dois programas quaisquer. Aqui está um exemplo que você certamente irá encontrar, de uma forma ou de outra:

```
$ ls /dsafsda
```

```
ls: cannot access /dsafsda: No such file or directory
```

Há três componentes nessa mensagem:

- O nome do programa, que é `ls`. Alguns programas omitem essa informação de identificação, o que pode ser irritante ao escrever shell scripts, porém não é um problema tão sério assim.
- O nome do arquivo, `/dsafsda`, que é uma informação mais específica. Há um problema com esse path.
- O erro `No such file or directory` (esse arquivo ou diretório não existe) indica o problema com o nome do arquivo.

Reunindo tudo, você terá algo como “`ls` tentou abrir `/dsafsda`, porém não pôde porque ele não existe”. Isso pode parecer óbvio, mas essas mensagens podem se tornar um pouco confusas quando um shell script que incluía um comando incorreto com um nome diferente é executado.

Ao tentar solucionar os problemas, sempre trate o primeiro erro antes. Alguns programas informam que não podem fazer algo antes de relatar uma série de outros problemas. Por exemplo, suponha que você execute um programa fictício chamado `scumd` e que a seguinte mensagem de erro tenha sido apresentada:

```
scumd: cannot access /etc/scumd/config: No such file or directory
```

Depois dessa informação, há uma lista enorme de outras mensagens de erro que parecem indicar uma total catástrofe. Não deixe que esses outros erros o distraiam. Provavelmente, você só precisará criar */etc/scumd/config*.

👍 **Observação:** não confunda mensagens de erro com mensagens de aviso (warnings). Os avisos geralmente se parecem com os erros, porém contêm a palavra *warning*. Um aviso normalmente indica que algo está errado, mas que o programa continuará tentando executar de qualquer maneira. Para corrigir um problema apontado em uma mensagem de aviso, talvez seja necessário ir atrás de um processo e matá-lo antes de executar qualquer outra tarefa. (Você aprenderá a listar e a matar processos na seção 2.16.)

2.15.2 Erros comuns

Muitos erros que você verá em programas Unix resultam de erros associados a arquivos e processos. A seguir, apresentamos um desfile de mensagens de erro:

No such file or directory (Esse arquivo ou diretório não existe)

Esse é o erro número um. Você tentou acessar um arquivo que não existe. Como o sistema de I/O de arquivos do Unix não faz discriminação entre arquivos e diretórios, essa mensagem de erro ocorre em todos os lugares. Você irá obtê-la ao tentar ler um arquivo que não exista, ao tentar mudar para um diretório que não esteja presente, ao tentar escrever em um arquivo em um diretório que não exista e assim por diante.

File exists (O arquivo existe)

Nesse caso, provavelmente você tentou criar um arquivo que já existe. Isso é comum ao tentar criar um diretório com o mesmo nome de um arquivo.

Not a directory, Is a directory (Não é um diretório, É um diretório)

Essas mensagens aparecem quando você tenta usar um arquivo como um diretório ou um diretório como um arquivo. Por exemplo:

```
$ touch a
```

```
$ touch a/b
```

```
touch: a/b: Not a directory
```

Observe que a mensagem de erro somente se aplica à parte referente a *a* de *a/b*. Ao se deparar com esse problema, talvez seja necessário explorar um pouco mais para descobrir qual é o componente do path que está sendo tratado como um diretório.

No space left on device (Não há espaço no dispositivo)

Seu espaço em disco acabou.

Permission denied (Permissão negada)

Esse erro será obtido se você tentar ler ou escrever em um arquivo ou em um

diretório para o qual você não tenha permissão de acesso (você não tem privilégios suficientes). Esse erro também aparecerá se você tentar executar um arquivo que não tenha o bit de execução ligado (mesmo que você possa ler o arquivo). Você lerá mais sobre permissões na seção 2.17.

Operation not permitted (Operação não permitida)

Esse erro normalmente acontece quando você tenta matar um processo que não seja seu.

Segmentation fault, Bus error (Falha de segmentação, Erro de barramento)

Uma *falha de segmentação* significa essencialmente que a pessoa que escreveu o programa que acabou de ser executado cometeu um erro em algum lugar. O programa tentou acessar uma parte da memória para a qual ele não tinha permissão de acessar, e o sistema operacional o matou. De modo semelhante, um *erro de barramento* significa que o programa tentou acessar uma parte da memória de uma determinada maneira que não deveria. Ao obter um desses erros, pode ser que você tenha fornecido algum dado de entrada que o programa não estava esperando.

2.16 Listando e manipulando processos

Lembre-se de que, de acordo com o capítulo 1, um *processo* é um programa em execução. Cada processo no sistema tem um *ID de processo* numérico (PID). Para uma listagem rápida dos processos em execução, basta executar `ps` na linha de comando. Você deverá obter uma lista como esta:

```
$ ps
PID TTY STAT TIME COMMAND
520 p0 S  0:00 -bash
545  ? S   3:59 /usr/X11R6/bin/ctwm -W
548  ? S   0:10 xclock -geometry -0-0
2159 pd SW  0:00 /usr/bin/vi lib/addresses
31956 p3 R   0:00 ps
```

Os campos estão descritos a seguir:

- PID – é o ID do processo.
- TTY – é o dispositivo de terminal em que o processo está executando. Outros detalhes sobre esse assunto estão mais adiante.
- STAT – é o status do processo, ou seja, o que o processo está fazendo e em que local está a sua memória. Por executando, s significa dormindo (sleeping) e R significa em execução (running). (Veja a página de manual `ps(1)` para obter uma

descrição de todos os símbolos.)

- **TIME** – é a quantidade de tempo de CPU em minutos e segundos que o processo usou até o momento. Em outras palavras, é a quantidade total de tempo gasto pelo processo executando instruções no processador.
- **COMMAND** – isso parece ser óbvio, mas saiba que um processo pode alterar esse campo em relação ao seu valor original.

2.16.1 Opções de comandos

O comando `ps` tem várias opções. Para deixar a situação mais confusa, é possível especificar as opções de três maneiras diferentes: Unix, BSD e GNU. Muitas pessoas acham que o estilo BSD é o mais confortável (talvez porque envolva menos digitação), portanto usaremos esse estilo neste livro. A seguir, estão algumas das combinações de opções mais úteis:

- `ps x` – mostra todos os processos em execução.
- `ps ax` – mostra todos os processos do sistema, e não apenas aqueles dos quais você é o proprietário.
- `ps u` – inclui informações mais detalhadas sobre os processos.
- `ps w` – mostra os nomes completos dos comandos, e não apenas o que cabe em uma linha.

Como ocorre com outros programas, as opções podem ser combinadas, conforme podemos ver em `ps aux` e `ps auxw`.

Para verificar um processo específico, adicione o seu PID na lista de argumentos do comando `ps`. Por exemplo, para inspecionar o processo do shell corrente, use `ps u $$`, pois `$$` é uma variável de shell avaliada como o PID do shell corrente. (Você encontrará informações sobre os comandos de administração `top` e `lsof` no capítulo 8. Eles podem ser úteis para localizar processos, mesmo quando você estiver fazendo algo que não seja manutenção do sistema.)

2.16.2 Matando processos

Para encerrar um processo, envie-lhe um *sinal* com o comando `kill`. Um sinal é uma mensagem do kernel para um processo. Ao executar `kill`, você estará pedindo ao kernel que envie um sinal para outro processo. Na maioria dos casos, tudo o que você precisará fazer é executar:

```
$ kill pid
```

Há vários tipos de sinais. O default é `TERM`, que quer dizer *terminate* (terminar). Diferentes sinais podem ser enviados por meio da adição de uma opção extra em `kill`. Por exemplo, para congelar um processo em vez de encerrá-lo, utilize o sinal `STOP`:

```
$ kill -STOP pid
```

Um processo interrompido continuará na memória, pronto para retomar a partir do ponto em que parou. Use o sinal `CONT` para continuar a execução do processo novamente:

```
$ kill -CONT pid
```

👍 **Observação:** usar `Ctrl-C` para encerrar um processo que está executando no terminal corrente é o mesmo que usar `kill` para terminar o processo usando o sinal `INT` (interrupt, ou interromper).

A maneira mais brutal de terminar um processo é usar o sinal `KILL`. Outros sinais dão ao processo uma chance de fazer uma limpeza, porém isso não ocorre com `KILL`. O sistema operacional encerra o processo e força a sua remoção da memória. Use-o como último recurso.

Não mate processos indiscriminadamente, em especial se não souber o que está fazendo.

Pode ser que você dê um tiro no próprio pé. Você poderá ver outros usuários fornecendo números no lugar de nomes com `kill`; por exemplo, `kill -9` no lugar de `kill -KILL`. Isso ocorre porque o kernel utiliza números para representar os diferentes sinais; você pode usar `kill` dessa maneira se souber o número do sinal que deseja enviar.

2.16.3 Controle de jobs

Os shells também suportam *controle de jobs* (job control), que é uma maneira de enviar os sinais `TSTP` (semelhante a `STOP`) e `CONT` para os programas usando diversas teclas e vários comandos. Por exemplo, um sinal `TSTP` pode ser enviado com `Ctrl-Z`; o processo pode ser iniciado novamente com `fg` (bring to foreground, ou trazer para primeiro plano) ou `bg` (move to background, ou mover para segundo plano) – veja a próxima seção. Porém, apesar de sua utilidade e dos hábitos de muitos usuários experientes, o controle de jobs não é necessário e pode ser confuso para os iniciantes: é comum os usuários pressionarem `Ctrl-Z` no lugar de `Ctrl-C`, esquecerem-se do que estavam executando e, posteriormente, acabarem ficando com vários processos suspensos.

👍 **Dica:** para ver se você não suspendeu acidentalmente algum processo em seu terminal corrente, execute o comando `jobs`.

Se quiser ter vários shells, execute cada programa em uma janela diferente de terminal, coloque os processos que não sejam interativos em background (conforme será

explicado na próxima seção) ou aprenda a usar o programa `screen`.

2.16.4 Processos em background

Normalmente, ao executar um comando Unix a partir do shell, você não terá o prompt de shell de volta até que o programa tenha acabado de executar. Entretanto um processo pode ser desassociado do shell e colocado em “background” com o uso de `&` (ampersand, ou “e comercial”); isso fará o prompt ser devolvido. Por exemplo, se você tiver um arquivo grande que deva ser descompactado com `gunzip` (você verá esse assunto na seção 2.18), e quiser executar outras tarefas enquanto isso estiver sendo feito, execute um comando como este:

```
$ gunzip arquivo.gz &
```

O shell deve responder exibindo o PID do novo processo em background, e o prompt deverá ser devolvido imediatamente para que você possa continuar trabalhando. O processo continuará a executar depois que você fizer logout, o que, particularmente, vem a calhar se for necessário executar um programa que faça bastante processamento numérico por um tempo. (Conforme a sua instalação, o shell poderá notificá-lo quando o processo terminar.)

A desvantagem de executar processos em background está no fato de que eles podem querer trabalhar com a entrada-padrão (ou, pior ainda, ler diretamente do terminal). Se um programa tentar ler algo da entrada-padrão enquanto estiver em background, ele poderá congelar (tente executar `fg` para trazê-lo de volta) ou terminar. Além do mais, se o programa escrever na saída-padrão ou no erro-padrão, a saída poderá aparecer na janela do terminal, sem se importar com qualquer outra tarefa que estiver executando ali, o que significa que você poderá obter uma saída inesperada quando estiver trabalhando com algo diferente.

A melhor maneira de garantir que um processo em background não irá importuná-lo é redirecionar a sua saída (e, possivelmente, a entrada), conforme descrito na seção 2.14.

Se uma saída espúria de processos em background entrar em seu caminho, aprenda a redesenhar o conteúdo de sua janela do terminal. O shell `bash` e a maioria dos programas interativos de tela cheia suportam `Ctrl-L` para redesenhar toda a tela. Se um programa estiver lendo da entrada-padrão, `Ctrl-R` normalmente redesenhará a linha corrente, porém teclar a sequência incorreta na hora errada poderá deixá-lo em uma situação pior que a anterior. Por exemplo, dar `Ctrl-R` no prompt do `bash` colocará você em modo `isearch` reverso (tecle `Esc` para sair).

2.17 Modos e permissões de arquivo

Todo arquivo Unix tem um conjunto de *permissões* que determina se você pode ler, escrever ou executar o arquivo. Executar `ls -l` faz com que as permissões sejam exibidas. Aqui está um exemplo do resultado de um comando como esse:

```
-rw-r--r--❶ juser somegroup 7041 Mar 26 19:34 endnotes.html
```

O *modo* do arquivo **❶** representa suas permissões e algumas informações extras. Há quatro partes no modo, conforme ilustrado na figura 2.1.

O primeiro caractere do modo corresponde ao *tipo do arquivo*. Um traço (-) nessa posição, como no exemplo, representa um arquivo *normal*, o que significa que não há nada de especial nele. De longe, é o tipo mais comum de arquivo. Os diretórios também são comuns e são indicados por um `d` na posição do tipo de arquivo. (A seção 3.1 lista os tipos de arquivo restantes.)

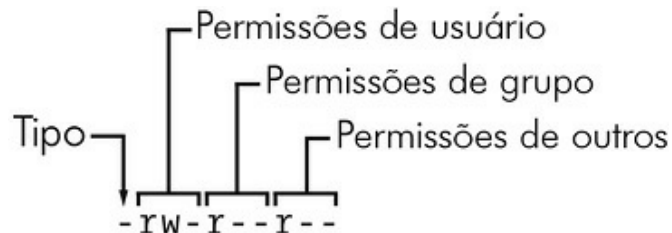


Figura 2.1 – As partes de um modo de arquivo.

O restante do modo de um arquivo contém as permissões, que podem ser separadas em três conjuntos: *usuário* (user), *grupo* (group) e *outros* (other), nessa ordem. Por exemplo, os caracteres `rw-` do exemplo correspondem às permissões de usuário, os caracteres `r--` que se seguem são as permissões de grupo e os caracteres `r--` finais são as permissões para outros.

Cada conjunto de permissões pode conter quatro representações básicas:

- `r` – indica que o arquivo pode ser lido.
- `w` – indica que o arquivo pode ser escrito.
- `x` – indica que o arquivo é executável (você pode executá-lo como um programa).
- `-` – não significa nada.

As permissões de usuário (o primeiro conjunto) dizem respeito ao usuário que é dono do arquivo. No exemplo anterior, esse usuário é `juser`. O segundo conjunto, que corresponde às permissões de grupo, diz respeito ao grupo do arquivo (`somegroup`, no exemplo). Qualquer usuário que estiver nesse grupo poderá tirar proveito dessas permissões. (Use o comando `groups` para ver em que grupos você está e consulte a seção

7.3.5 para obter mais informações.)

Todos os demais usuários do sistema terão acesso de acordo com o terceiro conjunto – as permissões para outros, que, às vezes, são chamadas de permissões *globais* (world permissions).

 **Observação:** cada posição de permissão referente à leitura, escrita e execução, às vezes, é chamada de *bit de permissão*. Desse modo, você poderá ouvir as pessoas se referirem a partes das permissões como “bits de leitura”.

Alguns arquivos executáveis têm um `s` listado nas permissões de usuário no lugar de um `x`. Isso indica que o executável é *setuid*, o que significa que, ao executar o programa, o dono do arquivo, e não você, será o usuário. Muitos programas usam esse bit de *setuid* para executar como root a fim de obter os privilégios necessários para alterar arquivos de sistema. Um exemplo é o programa `passwd`, que deve alterar o arquivo `/etc/passwd`.

2.17.1 Modificando as permissões

Para alterar as permissões, utilize o comando `chmod`. Inicialmente, selecione o conjunto de permissões que você quer mudar e, em seguida, o bit a ser alterado. Por exemplo, para adicionar permissões de leitura para grupo (`g`) e para outros (`o`, de “other”) em *arquivo*, estes dois comandos podem ser executados:

```
$ chmod g+r arquivo
```

```
$ chmod o+r arquivo
```

Ou faça tudo em um só passo:

```
$ chmod go+r arquivo
```

Para remover essas permissões, use `go-r` no lugar de `go+r`.

 **Observação:** obviamente, você não deve permitir que os arquivos possam ser globalmente escritos, pois fazer isso dará capacidade a qualquer pessoa em seu sistema de alterá-los. Contudo isso permitirá que qualquer pessoa conectada a Internet altere seus arquivos? Provavelmente não, a menos que o seu sistema tenha uma brecha de segurança de rede. Nesse caso, as permissões de arquivo não ajudarão, de qualquer maneira.

Às vezes, você poderá ver pessoas alterando permissões usando números, por exemplo:

```
$ chmod 644 arquivo
```

Essa alteração é chamada de *absoluta*, pois ela configura todos os bits de permissão de uma só vez. Para entender o seu funcionamento, você deve saber como representar os bits de permissão em formato octal (cada número está representado em base 8 e corresponde a um conjunto de permissões). Consulte a página de manual `chmod(1)` ou o manual `info` para obter mais informações.

Você não precisa realmente saber como compor modos absolutos; basta memorizar os modos usados com mais frequência. A tabela 2.4 lista os modos mais comuns.

Tabela 2.4 – Modos de permissão absolutos

Modo	Significado	Usado para
644	usuário: leitura/escrita; grupo, outros: leitura	arquivos
600	usuário: leitura/escrita; grupo, outros: nenhum	arquivos
755	usuário: leitura/escrita/execução; grupo, outros: leitura/execução	diretórios, programas
700	usuário: leitura/escrita/execução; grupo, outros: nenhum	diretórios, programas
711	usuário: leitura/escrita/execução; grupo, outros: execução	diretórios

Os diretórios também têm permissões. O conteúdo de um diretório pode ser listado se puder ser lido, porém você só poderá acessar um arquivo em um diretório se o diretório for executável. (Um erro comum que as pessoas cometem ao definir permissões de diretórios é remover acidentalmente a permissão de execução ao usar modos absolutos.)

Por fim, você pode especificar um conjunto de permissões default usando o comando de shell `umask`, que aplica um conjunto predefinido de permissões a qualquer arquivo novo que você criar. Em geral, use `umask 022` se quiser que todos possam ver os arquivos e diretórios que você criar, e use `umask 077` se não quiser. (Você deverá colocar o comando `umask` com o modo desejado em um dos seus arquivos de inicialização para que suas novas permissões default se apliquem às sessões seguintes, conforme será discutido no capítulo 13.)

2.17.2 Links simbólicos

Um *link simbólico* é um arquivo que aponta para outro arquivo ou um diretório, criando efetivamente um alias (como um atalho no Windows). Os links simbólicos proporcionam um acesso rápido a paths de diretório obscuros.

Em uma listagem longa de diretório, os links simbólicos têm o seguinte aspecto (observe o `l` como o tipo de arquivo no modo):

```
lrwxrwxrwx 1 ruser users 11 Feb 27 13:52 somedir -> /home/origdir
```

Se você tentar acessar *somedir* nesse diretório, o sistema lhe dará */home/origdir* em seu lugar. Os links simbólicos são apenas nomes que apontam para outros nomes. Seus nomes e os paths para os quais eles apontam não precisam significar nada. Por exemplo, */home/origdir* não precisa nem mesmo existir.

De fato, se */home/origdir* não existir, qualquer programa que acessar *somedir* informará que *somedir* não existe (exceto para `ls somedir`, um comando que, de forma estúpida, informa que *somedir* é *somedir*). Isso pode ser surpreendente, pois você pode ver algo chamado *somedir* bem diante de seus olhos.

Essa não é a única maneira pela qual os links simbólicos podem ser confusos. Outro problema é que você não poderá identificar as características do alvo de um link simplesmente olhando para o nome desse link; você deverá segui-lo para ver se ele aponta para um arquivo ou um diretório. Seu sistema também pode ter links que apontem para outros links – são chamados de *links simbólicos encadeados*.

2.17.3 Criando links simbólicos

Para criar um link simbólico de *alvo* para *nomedolink*, use `ln -s`:

```
$ ln -s alvo nomedolink
```

O argumento *nomedolink* corresponde ao nome do link simbólico, o argumento *alvo* é o path do arquivo ou do diretório para o qual o link *aponta* e a flag `-s` especifica um link simbólico (veja o aviso mais adiante).

Ao criar um link simbólico, verifique o comando duas vezes antes de executá-lo, pois muitos erros podem ocorrer. Por exemplo, se a ordem dos argumentos for invertida (`ln -s nomedolink alvo`), você terá um pouco de diversão se *nomedolink* for um diretório já existente. Se esse for o caso (e, com muita frequência, é), `ln` criará um link chamado *alvo* dentro de *nomedolink*, e o link apontará para si mesmo, a menos que *nomedolink* seja um path completo. Se algo der errado na criação de um link simbólico para um diretório, verifique se não há links simbólicos perdidos nesse diretório e remova-os.

Os links simbólicos também podem causar dores de cabeça se você não souber que eles existem. Por exemplo, você pode facilmente editar o que você pensa ser uma cópia de um arquivo, mas que, na realidade, é um link simbólico para o original.

 **AVISO:** não se esqueça da opção `-s` ao criar um link simbólico. Sem ele, `ln` criará um hard link e dará um nome de arquivo adicional a um único arquivo. O novo nome de arquivo terá o status do antigo; ele apontará (fará o link) diretamente para os dados do arquivo em vez de fazê-lo para outro nome de arquivo, como ocorre com um link simbólico. Os hard links podem ser mais confusos ainda que os links simbólicos. A menos que você compreenda o material que está na seção 4.5, evite usá-los.

Com todos esses avisos relacionados aos links simbólicos, por que alguém se incomodaria em usá-los? Porque eles oferecem uma maneira conveniente de organizar e compartilhar arquivos, além de servirem para corrigir pequenos problemas.

2.18 Arquivamento e compressão de arquivos

Agora que você já aprendeu sobre arquivos, permissões e possíveis erros, você deverá dominar o `gzip` e o `tar`.

2.18.1 `gzip`

O programa `gzip` (GNU Zip) é um dos padrões atuais de programas de compressão do Unix. Um arquivo que termine com `.gz` é um arquivo GNU Zip. Use `gunzip arquivo.gz` para descompactar `<arquivo>.gz` e remover o sufixo; para compactá-lo novamente, use `gzip arquivo`.

2.18.2 `tar`

De modo diferente dos programas `zip` para outros sistemas operacionais, o `gzip` não faz arquivamento, ou seja, ele não empacota vários arquivos e diretórios em um só arquivo. Para fazer um arquivamento (criar um archive), use `tar`:

```
$ tar cvf arquivo.tar arquivo1 arquivo2 ...
```

Os arquivos criados pelo `tar` normalmente têm um sufixo `.tar` (é uma convenção – não é obrigatório). Por exemplo, no comando anterior, `arquivo1`, `arquivo2` e assim por diante correspondem aos nomes dos arquivos e diretórios que você deseja arquivar em `<arquivo>.tar`. A flag `c` ativa o *modo de criação* (create mode). As flags `v` e `f` têm funções mais específicas.

A flag `v` ativa a saída mais extensa para diagnóstico, fazendo com que `tar` mostre os nomes dos arquivos e dos diretórios quando encontrá-los no arquivo `tar`. Adicionar outro `v` faz com que `tar` mostre detalhes como o tamanho e as permissões dos arquivos. Se não quiser que o `tar` informe o que ele estiver fazendo, omita a flag `v`.

A flag `f` indica a opção de arquivo. O próximo argumento na linha de comando após a flag `f` deve ser o arquivo a ser criado pelo `tar` (no exemplo anterior, é `<arquivo>.tar`). Use sempre essa opção seguida do nome de um arquivo, exceto com drives de fita. Para usar a entrada ou a saída-padrão, digite um traço (-) no lugar do nome do arquivo.

Desempacotando arquivos `tar`

Para desempacotar um arquivo `.tar` com `tar`, use a flag `x`:

```
$ tar xvf arquivo.tar
```

Nesse comando, a flag `x` coloca `tar` em *modo de extração* (desempacotamento). Você pode extrair partes individuais do arquivo ao inserir os nomes das partes no final da linha de comando, porém seus nomes exatos deverão ser conhecidos. [Para ter certeza dos nomes, veja o modo tabela de conteúdo (table-of-contents mode) descrito mais

adiante].

 **Observação:** ao usar o modo de extração, lembre-se de que tar não apaga o arquivo *.tar* após extrair o seu conteúdo.

Modo tabela de conteúdo

Antes de desempacotar um arquivo *.tar*, normalmente é uma boa ideia verificar o seu conteúdo usando o *modo tabela de conteúdo* (table-of-contents mode) usando a flag *t* no lugar da flag *x*. Esse modo verifica a integridade básica do arquivo tar e exibe os nomes de todos os arquivos nele contidos. Se um arquivo tar não for testado antes de ser desempacotado, você poderá acabar descarregando uma quantidade enorme de arquivos confusos no diretório corrente, que poderá ser bem difícil de limpar.

Ao conferir um arquivo tar com o modo *t*, verifique se os itens estão em uma estrutura racional de diretório, ou seja, se todos os nomes de paths dos arquivos começam com o mesmo diretório. Se você não estiver seguro, crie um diretório temporário, vá para esse diretório e, em seguida, faça a extração. (É sempre possível usar *mv * ..* se o arquivo tar não criar confusão.)

Ao desempacotar o arquivo, considere o uso da opção *p* para preservar as permissões. Use-a em modo de extração para sobrescrever o seu *umask* e obter as permissões exatas especificadas no arquivo tar. A opção *p* é default ao trabalhar como superusuário. Se estiver tendo problemas com as permissões e as propriedades ao desempacotar um arquivo tar como superusuário, certifique-se de estar esperando o comando terminar e o prompt de shell ser obtido novamente. Embora você possa querer extrair somente uma pequena parte de um arquivo, tar deverá percorrer tudo e você não deverá interromper o processo, pois as permissões serão configuradas somente *após* a verificação completa do arquivo tar.

Guarde *todas* as opções e os modos do tar descritos nesta seção em sua memória. Se estiver tendo problemas, anote-os em cartões para auxiliar na memorização. Pode parecer uma atividade da época do ensino fundamental, porém é muito importante evitar erros em decorrência de descuidos ao usar esse comando.

2.18.3 Arquivos compactados (*.tar.gz*)

Muitos iniciantes acham confuso o fato de os arquivos tar se encontrarem normalmente compactados, com nomes de arquivo terminados com *.tar.gz*. Para desempacotar um arquivo compactado, trabalhe da direita para a esquerda; livre-se primeiro do *.gz* e, em seguida, preocupe-se com o *.tar*. Por exemplo, os dois comandos a seguir descompactam e desempacotam *<arquivo>.tar.gz*:

```
$ gunzip arquivo.tar.gz
```

```
$ tar xvf arquivo.tar
```

Quando estiver começando a aprender, você poderá fazer isso, um passo de cada vez, executando inicialmente o `gunzip` para descompactar e, em seguida, `tar` para efetuar uma verificação e o desempacotamento. Para criar um arquivo `tar` compactado, faça o inverso: execute `tar` antes, e depois `gzip`. Faça isso com bastante frequência e, logo, você irá memorizar como os processos de arquivamento e de compressão funcionam. Você também irá se cansar de toda essa digitação e começará a procurar atalhos. Vamos dar uma olhada neles agora.

2.18.4 zcat

O método mostrado anteriormente não é a maneira mais rápida nem mais eficiente de chamar `tar` em um arquivo compactado e desperdiça espaço em disco e tempo de I/O do kernel. Uma maneira melhor consiste em combinar as funções de arquivamento e de compressão usando um pipeline. Por exemplo, esse pipeline de comandos descompacta `<arquivo>.tar.gz`:

```
$ zcat arquivo.tar.gz | tar xvf -
```

O comando `zcat` é igual a `gunzip -dc`. A opção `-d` descompacta e a opção `-c` envia o resultado para a saída-padrão (nesse caso, para o comando `tar`).

Pelo fato de usar `zcat` ser tão comum, a versão de `tar` que vem com o Linux tem um atalho. Você pode usar `z` como opção para chamar `gzip` automaticamente no arquivo; isso funciona tanto para a extração (com os modos `x` ou `t` no `tar`) quanto para a criação (com `c`) de um arquivo. Por exemplo, use o comando a seguir para verificar um arquivo compactado:

```
$ tar ztvf arquivo.tar.gz
```

No entanto você deve tentar dominar o formato mais longo antes de usar o atalho.

 **Observação:** um arquivo `.tgz` é igual a um arquivo `.tar.gz`. O sufixo foi concebido para se adequar aos sistemas de arquivo FAT (baseados em MS-DOS).

2.18.5 Outros utilitários para compressão

Outro programa de compressão do Unix é o `bzip2`, cujos arquivos compactados terminam com `.bz2`. Embora seja um pouco mais lento que o `gzip`, o `bzip2` geralmente compacta um pouco melhor os arquivos-texto e, desse modo, está se tornando cada vez mais popular na distribuição de códigos-fonte. O programa de descompactação a ser usado é o `bunzip2`, e as opções de ambos os componentes são muito semelhantes às do `gzip` de modo que

não será necessário aprender nada novo. A opção de compactação/descompactação do bzip2 para o tar é j.

Um novo programa de compactação chamado xz também está ganhando popularidade. O programa correspondente de descompactação é o unxz, e os argumentos são semelhantes àqueles usados com gzip.

A maioria das distribuições Linux vem com os programas zip e unzip, que são compatíveis com os arquivos zip dos sistemas Windows. Eles funcionam nos arquivos .zip normais, assim como em arquivos com extração automática, terminados com .exe. Porém, se encontrar um arquivo que termine com .Z, você terá se deparado com uma relíquia criada pelo programa compress, que já foi padrão no Unix. O programa gunzip é capaz de descompactar esses arquivos, porém o gzip não os criará.

2.19 Aspectos essenciais da hierarquia de diretórios do Linux

Agora que você já sabe analisar arquivos, alterar diretórios e ler as páginas de manual, estará pronto para começar a explorar seus arquivos de sistema. Os detalhes da estrutura de diretórios do Linux estão apresentados no FHS (Filesystem Hierarchy Standard, ou Padrão para sistema de arquivos hierárquico, em <http://www.pathname.com/fhs/>), porém uma breve descrição deverá ser suficiente por enquanto.

A figura 2.2 oferece uma visão geral simplificada da hierarquia, mostrando alguns dos diretórios em /, /usr e /var. Observe que a estrutura de diretórios abaixo de /usr contém alguns dos mesmos nomes de diretório de /.

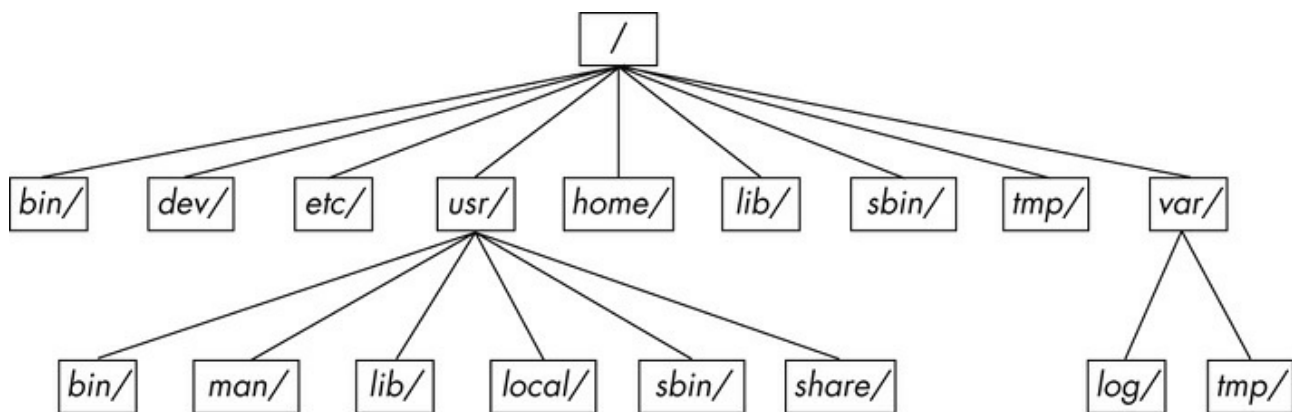


Figura 2.2 – Hierarquia de diretórios do Linux.

A seguir estão listados os subdiretórios mais importantes do diretório-raiz:

- /bin – contém programas prontos para execução (também conhecidos como

executáveis), incluindo a maior parte dos comandos Unix básicos como `ls` e `cp`. A maioria dos programas em `/bin` está no formato binário e foram criados por um compilador C, porém alguns são shell scripts em sistemas modernos.

- `/dev` – contém arquivos de dispositivos. Você aprenderá mais sobre eles no capítulo 3.
- `/etc` – esse diretório essencial de configurações do sistema contém arquivos de configuração de senha de usuário, de inicialização, de dispositivos, de rede e outros arquivos. Muitos itens em `/etc` são específicos do hardware do computador. Por exemplo, o diretório `/etc/X11` contém configurações da placa gráfica e de sistemas de janelas.
- `/home` – contém diretórios pessoais para usuários normais. A maioria das instalações Unix está em conformidade com esse padrão.
- `/lib` – é a abreviatura de library (biblioteca); esse diretório armazena arquivos de biblioteca contendo códigos que os executáveis podem usar. Há dois tipos de biblioteca: estática e compartilhada. O diretório `/lib` deve conter somente as bibliotecas compartilhadas, porém outros diretórios `lib`, como `/usr/lib`, contêm ambas as variedades, além de outros arquivos auxiliares. (Discutiremos as bibliotecas compartilhadas com mais detalhes no capítulo 15.)
- `/proc` – disponibiliza estatísticas do sistema por meio de uma interface de diretórios e de arquivos em que é possível navegar. Boa parte da estrutura de subdiretórios de `/proc` no Linux é única, porém muitas outras variantes de Unix têm recursos semelhantes. O diretório `/proc` contém informações sobre os processos em execução no momento, além de alguns parâmetros do kernel.
- `/sys` – esse diretório é semelhante a `/proc` no sentido em que oferece uma interface de dispositivo e de sistema. Você lerá mais sobre `/sys` no capítulo 3.
- `/sbin` – é o local em que estão os executáveis do sistema. Os programas nos diretórios `/sbin` estão relacionados ao gerenciamento de sistema, portanto usuários normais não têm componentes `/sbin` nos paths de seus comandos. Muitos dos utilitários encontrados aqui não funcionarão se não forem executados como root.
- `/tmp` – é uma área de armazenamento para arquivos menores e temporários com os quais você não se importe muito. Qualquer usuário poderá ler e escrever em `/tmp`, porém um usuário poderá não ter permissão para acessar arquivos de outros usuários nesse local. Muitos programas usam esse diretório como área de trabalho. Se alguma informação for extremamente importante, não a coloque em `/tmp`, pois a maioria das distribuições limpa esse diretório quando o computador é inicializado e

algumas até mesmo removem os arquivos antigos periodicamente. Além disso, não deixe */tmp* ficar repleto de lixo porque seu espaço normalmente será compartilhado com algo crítico (como o restante de */*, por exemplo).

- */usr* – embora pronunciado como “user”, esse subdiretório não tem nenhum arquivo de usuário. Em vez disso, ele contém uma hierarquia extensa de diretórios, incluindo a maior parte do sistema Linux. Muitos dos nomes de diretório em */usr* são iguais àqueles do diretório-raiz (como */usr/bin* e */usr/lib*) e contêm os mesmos tipos de arquivo. (O motivo de o diretório-raiz não conter o sistema completo é principalmente histórico – no passado, servia para manter os requisitos de espaço reduzidos.)
- */var* – é o subdiretório de variáveis, em que os programas registram informações de tempo de execução. O logging do sistema, os arquivos de monitoração de usuário, as caches e outros arquivos que os programas de sistema criam e administram estão aqui. (Você perceberá que há um diretório */var/tmp* aqui, porém o sistema não o limpa na inicialização.)

2.19.1 Outros subdiretórios do diretório-raiz

Há mais alguns subdiretórios interessantes no diretório-raiz:

- */boot* – contém os arquivos de boot loader do kernel. Esses arquivos dizem respeito somente ao primeiro estágio do procedimento de inicialização do Linux; você não encontrará informações sobre como o Linux inicializa seus serviços nesse diretório. Veja o capítulo 5 para obter mais informações sobre esse assunto.
- */media* – um ponto de associação base para mídias removíveis, por exemplo, pen drives, encontrado em várias distribuições.
- */opt* – esse diretório pode conter softwares adicionais de terceiros. Muitos sistemas não usam */opt*.

2.19.2 O diretório */usr*

O diretório */usr* pode parecer relativamente limpo à primeira vista, porém uma olhada rápida em */usr/bin* e em */usr/lib* revela que há muitos itens aqui; */usr* é o local em que a maioria dos programas e dados do espaço de usuário reside. Além de */usr/bin*, */usr/sbin* e */usr/lib*, */usr* contém o seguinte:

- */include* – armazena arquivos de header usados pelo compilador C.
- */info* – contém manuais de info GNU (veja a seção 2.13).

- */local* – é o local em que os administradores podem instalar seus próprios softwares. Sua estrutura deve ser semelhante à estrutura de */* e de */usr*.
- */man* – contém as páginas de manual.
- */share* – contém arquivos que devem funcionar em outros tipos de computadores Unix, sem perda de funcionalidades. No passado, as redes de computadores compartilhavam esse diretório, porém um verdadeiro diretório */share* está se tornando raro, pois não há problemas de espaço nos discos modernos. Manter um diretório */share*, em geral, significa apenas problemas. De qualquer maneira, */man*, */info* e outros subdiretórios geralmente são encontrados aqui.

2.19.3 Localização do kernel

Em sistemas Linux, o kernel normalmente está em */vmlinuz* ou em */boot/vmlinuz*. Um *boot loader* (carregador de boot) carrega esse arquivo na memória e o coloca em ação quando o sistema é inicializado. (Você verá detalhes sobre o boot loader no capítulo 5.)

Depois que o boot loader executar e colocar o kernel em ação, o arquivo principal do kernel não será mais usado pelo sistema em execução. Entretanto você encontrará muitos módulos que o kernel pode carregar e descarregar por demanda durante o curso normal de operação do sistema. Eles são chamados de *módulos carregáveis do kernel* e estão localizados em */lib/modules*.

2.20 Executando comandos como superusuário

Antes de prosseguir, você deve aprender a executar comandos como superusuário. Provavelmente, você já sabe que pode executar o comando `su` e fornecer a senha de root para iniciar um root shell. Essa prática funciona, porém tem algumas desvantagens:

- Você não terá nenhum registro de comandos que alterem o sistema.
- Você não terá nenhum registro dos usuários que realizaram comandos que alterem o sistema.
- Você não terá acesso ao seu ambiente normal de shell.
- Será preciso fornecer a senha de root.

2.20.1 sudo

A maioria das principais distribuições usa um pacote chamado `sudo` para permitir aos administradores executar comandos como root quando estiverem logados como eles mesmos. Por exemplo, no capítulo 7, você aprenderá a usar `vim` para editar o arquivo

/etc/passwd. Isso pode ser feito da seguinte maneira:

```
$ sudo vipw
```

Ao executar esse comando, `sudo` fará o log dessa ação com o serviço `syslog` em `local2`. Você também aprenderá mais sobre logs do sistema no capítulo 7.

2.20.2 */etc/sudoers*

É claro que o sistema não deixa simplesmente *qualquer* usuário executar comandos como superusuário; você deve configurar os usuários privilegiados em seu arquivo */etc/sudoers*. O pacote `sudo` tem várias opções (que provavelmente, você jamais usará), tornando a sintaxe de */etc/sudoers* um tanto quanto complicada. Por exemplo, este arquivo confere a *user1* e a *user2* a capacidade de executar qualquer comando como `root`, sem a necessidade de fornecer uma senha:

```
User_Alias ADMINS = user1, user2
```

```
ADMINS ALL = NOPASSWD: ALL
```

```
root ALL=(ALL) ALL
```

A primeira linha define um alias de usuário `ADMINS` para os dois usuários; e a segunda linha concede os privilégios. A parte referente a `ALL = NOPASSWD: ALL` significa que os usuários no alias `ADMINS` podem usar `sudo` para executar comandos como `root`. O segundo `ALL` quer dizer “qualquer comando”. O primeiro `ALL` quer dizer “qualquer host”. (Se você tiver mais de um computador, será possível definir tipos diferentes de acesso para cada computador ou grupo de computadores, porém não discutiremos essa funcionalidade.)

`root ALL=(ALL) ALL` simplesmente quer dizer que o superusuário também pode usar `sudo` para executar qualquer comando em qualquer host. O `(ALL)` extra quer dizer que o superusuário também pode executar comandos como qualquer outro usuário. Esse privilégio poderá ser estendido aos usuários `ADMINS` ao adicionar `(ALL)` à linha de */etc/sudoers*, como mostrado em ❶:

```
ADMINS ALL = (ALL)❶ NOPASSWD: ALL
```

 **Observação:** use o comando `visudo` para editar */etc/sudoers*. Esse comando verifica se há erros de sintaxe no arquivo depois que ele for salvo.

Por enquanto, é o que temos a dizer sobre `sudo`. Se for necessário usar seus recursos mais avançados, consulte as páginas de manual `sudoers(5)` e `sudo(8)`. (O verdadeiro funcionamento da mudança de usuário será discutido no capítulo 7.)

2.21 Próximos passos

Agora você já deve saber como fazer o seguinte na linha de comando: executar programas, redirecionar saídas, interagir com arquivos e diretórios, ver listagens de processos, visualizar páginas de manual e, em geral, reconhecer o espaço de usuário de um sistema Linux. Você também deverá ser capaz de executar comandos como superusuário. Talvez você ainda não saiba muito sobre os detalhes internos dos componentes do espaço de usuário ou o que acontece no kernel, porém, de posse do básico sobre arquivos e processos, você estará no caminho certo. Nos próximos capítulos, iremos trabalhar com os componentes tanto de sistema do kernel quanto do espaço de usuário, utilizando as ferramentas de linha de comando que você acabou de conhecer.

¹ N.T.: glob é um termo utilizado no contexto de programação de computadores para descrever uma forma de casamento de padrões (fonte: [http://pt.wikipedia.org/wiki/Glob_\(programação\)](http://pt.wikipedia.org/wiki/Glob_(programação)))).

CAPÍTULO 3

Dispositivos

Este capítulo contém um tour básico pela infraestrutura de dispositivos disponibilizada pelo kernel em um sistema Linux em operação. Ao longo da história do Linux, houve várias mudanças em relação ao modo como o kernel apresenta os dispositivos ao usuário. Começaremos dando uma olhada no sistema tradicional de arquivos de dispositivo para ver como o kernel disponibiliza informações de configuração de dispositivos por meio de sysfs. Nosso objetivo é ser capaz de extrair informações sobre os dispositivos em um sistema a fim de entender algumas operações rudimentares. A interação com tipos específicos de dispositivo será discutida com mais detalhes em capítulos posteriores.

É importante entender como o kernel interage com o espaço de usuário quando estiver diante de novos dispositivos. O sistema udev permite que os programas no espaço de usuário configurem automaticamente e usem novos dispositivos. Você verá o funcionamento básico do envio de uma mensagem do kernel para um processo no espaço de usuário por meio do udev e o que o processo faz com essa mensagem.

3.1 Arquivos de dispositivo

É fácil manipular a maioria dos dispositivos em um sistema Unix porque o kernel apresenta várias das interfaces de I/O dos dispositivos aos processos de usuário na forma de arquivos. Esses arquivos de dispositivo às vezes são chamados de *nós de dispositivo* (device nodes). Um programador não só pode usar operações normais de arquivo para trabalhar com um dispositivo como também alguns dispositivos são acessíveis a programas-padrão como `cat`, portanto não é necessário ser um programador para usar um dispositivo. Entretanto há um limite para o que você pode fazer com uma interface de arquivo, de modo que nem todos os dispositivos ou recursos de dispositivos serão acessíveis por meio de I/O padrão de arquivo.

O Linux usa o mesmo design utilizado por outras variantes de Unix para arquivos de dispositivo. Os arquivos de dispositivo estão no diretório `/dev`; executar `ls /dev` mostrará vários arquivos em `/dev`. Como podemos trabalhar com os dispositivos?

Para começar, considere o comando a seguir:

```
$ echo blah blah > /dev/null
```

Como ocorre com qualquer comando cuja saída é redirecionada, esse comando envia informações da saída-padrão para um arquivo. No entanto o arquivo é */dev/null*, que é um dispositivo, e o kernel decide o que fará com qualquer dado escrito nesse dispositivo. No caso de */dev/null*, o kernel simplesmente irá ignorar a entrada e jogará os dados fora.

Para identificar um dispositivo e visualizar suas permissões, use `ls -l`:

Listagem 3.1 – Arquivos de dispositivo

```
$ ls -l
```

```
brw-rw---- 1 root disk 8, 1 Sep  6 08:37 sda1
crw-rw-rw- 1 root root 1, 3 Sep  6 08:37 null
prw-r--r-- 1 root root  0 Mar  3 19:17 fdata
srw-rw-rw- 1 root root  0 Dec 18 07:43 log
```

Observe o primeiro caractere de cada linha (o primeiro caractere do modo do arquivo) na listagem 3.1. Se esse caractere for *b*, *c*, *p* ou *s*, o arquivo será um dispositivo. Essas letras significam *block* (bloco), *character* (caractere), *pipe* e *socket*, respectivamente, conforme descrito com mais detalhes a seguir.

Dispositivo de bloco

Os programas acessam dados de um *dispositivo de bloco* (block device) em porções fixas. *sda1* no exemplo anterior é um *dispositivo de disco* – um tipo de dispositivo de bloco. Os discos podem ser facilmente divididos em blocos de dados. Como o tamanho total de um dispositivo de bloco é fixo e fácil de indexar, os processos podem fazer acessos aleatórios a qualquer bloco do dispositivo com a ajuda do kernel.

Dispositivo de caractere

Os dispositivos de caractere funcionam com streams de dados. Você pode somente ler ou escrever caracteres em dispositivos de caractere, conforme demonstrado anteriormente com */dev/null*. Os dispositivos de caractere não têm um tamanho; quando você lê de um desses dispositivos ou escreve em um, o kernel normalmente realiza uma operação de leitura ou de escrita no dispositivo. As impressoras ligadas diretamente ao seu computador são representadas por dispositivos de caractere. É importante observar que, durante uma interação com um dispositivo de caractere, o kernel não poderá fazer backup nem analisar novamente o stream de dados depois que esses tiverem sido passados para um dispositivo ou um processo.

Dispositivo de pipe

Pipes nomeados (named pipes) são como dispositivos de caractere, com um processo diferente na outra extremidade do stream de I/O no lugar de um driver do kernel.

Dispositivo de socket

Sockets são interfaces para propósitos especiais, frequentemente usados para comunicação entre processos. Geralmente, eles são encontrados fora do diretório */dev*. Os arquivos de socket representam sockets do domínio Unix; você aprenderá mais sobre eles no capítulo 10.

Os números antes das datas nas duas primeiras linhas da listagem 3.1 correspondem aos números *principal* (major) e *secundário* (minor) dos dispositivos, que ajudam o kernel a identificar o dispositivo. Dispositivos semelhantes normalmente têm o mesmo número principal, por exemplo, *sda3* e *sdb1* (ambos correspondem a partições de disco rígido).

 **Observação:** nem todos os dispositivos têm arquivos de dispositivo, pois as interfaces de I/O de dispositivos de bloco e de caractere não são apropriadas em todos os casos. Por exemplo, as interfaces de rede não têm arquivos de dispositivo. Teoricamente, seria possível interagir com uma interface de rede usando um único dispositivo de caractere, porém, como isso seria excepcionalmente difícil, o kernel utiliza outras interfaces de I/O.

3.2 Path de dispositivos sysfs

O diretório */dev* tradicional do Unix é uma maneira conveniente de os processos de usuário fazerem referência e terem uma interface com os dispositivos suportados pelo kernel, porém é também um esquema excessivamente simplista. O nome do dispositivo em */dev* diz um pouco sobre o dispositivo, mas não muito. Outro problema é que o kernel atribui os dispositivos na ordem em que eles são encontrados, portanto um dispositivo poderá ter um nome diferente entre diferentes reinicializações.

Para oferecer uma visão uniforme dos dispositivos conectados de acordo com seus verdadeiros atributos de hardware, o kernel do Linux oferece a interface sysfs por meio de um sistema de arquivos e de diretórios. O path base para os dispositivos é */sys/devices*. Por exemplo, o disco rígido SATA em */dev/sda* deve ter o seguinte path no sysfs:

```
/sys/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0:0/block/sda
```

Como você pode ver, esse path é bem longo quando comparado ao nome de arquivo */dev/sda*, e ele também é um diretório. Porém você não pode realmente comparar os dois paths porque eles têm finalidades diferentes. O arquivo */dev* existe para que os processos de usuário possam usar o dispositivo, enquanto o path */sys/devices* é usado

para visualizar informações e administrar o dispositivo. Ao listar o conteúdo de um path de dispositivo como o anterior, você verá algo semelhante a:

```
alignment_offset discard_alignment holders removable size uevent
bdi          events          inflight ro      slaves
capability   events_async   power   sda1    stat
dev          events_poll_msecs queue   sda2    subsystem
device       ext_range     range   sda5    trace
```

Os arquivos e os subdiretórios, nesse caso, foram criados principalmente para serem lidos por programas, e não por seres humanos, porém você pode ter uma ideia do que eles contêm e representam ao ver um exemplo como o arquivo */dev*. A execução de `cat dev` nesse diretório exibe os números 8:0, que são os números de dispositivo principal e secundário de */dev/sda*.

Há alguns atalhos no diretório */sys*. Por exemplo, */sys/block* deve conter todos os dispositivos de bloco disponíveis em um sistema. Entretanto eles são apenas links simbólicos; execute `ls -l /sys/block` para mostrar os verdadeiros paths de sysfs.

Pode ser difícil encontrar a localização sysfs de um dispositivo em */dev*. Utilize o comando `udevadm` para mostrar o path e outros atributos:

```
$ udevadm info --query=all --name=/dev/sda
```

👉 **Observação:** o programa `udevadm` está em */sbin*; você pode colocar esse diretório no final de seu path caso ele ainda não esteja lá.

Mais detalhes sobre `udevadm` e todo o sistema `udev` podem ser encontrados na seção 3.5.

3.3 dd e dispositivos

O programa `dd` é extremamente útil quando trabalhamos com dispositivos de bloco e de caractere. A única função desse programa é ler de um arquivo ou de um stream de entrada e escrever em um arquivo ou stream de saída, possivelmente fazendo alguma conversão de codificação no caminho.

`dd` copia dados em blocos de tamanho fixo. Eis o modo de usar `dd` com um dispositivo de caractere e algumas opções comuns:

```
$ dd if=/dev/zero of=new_file bs=1024 count=1
```

Como você pode ver, o formato das opções de `dd` difere dos formatos de opções da maioria dos outros comandos Unix; ele é baseado em um antigo estilo de JCL (Job Control Language, ou Linguagem de controle de jobs) da IBM. Em vez de usar o caractere hífen (-) para sinalizar uma opção, você deve nomeá-la e definir seu valor usando o sinal de igualdade (=). O exemplo anterior copia um único bloco de 1024

bytes de */dev/zero* (um stream contínuo de bytes zero) para *new_file*.

Estas são as opções importantes de *dd*:

- *if=arquivo* — é o arquivo de entrada. O default é a entrada-padrão.
- *of=arquivo* — é o arquivo de saída. O default é a saída-padrão.
- *bs=tamanho* — é o tamanho do bloco. *dd* lê e escreve essa quantidade de bytes de dados a cada vez. Para abreviar grandes porções de dados, você pode usar *b* e *k* para indicar 512 e 1024 bytes, respectivamente. Desse modo, o exemplo anterior poderia ter sido escrito como *bs=1k* em vez de *bs=1024*.
- *ibs=tamanho*, *obs=tamanho* — são os tamanhos dos blocos de entrada e de saída. Se você puder usar o mesmo tamanho de bloco tanto para a entrada quanto para a saída, use a opção *bs*; caso contrário, use *ibs* e *obs* para a entrada e para a saída, respectivamente.
- *count=num* — é o número total de blocos a serem copiados. Ao trabalhar com um arquivo enorme — ou com um dispositivo que forneça um stream de dados que não tenha fim, por exemplo, */dev/zero* —, você vai querer que *dd* pare em um determinado ponto; do contrário, muito espaço de disco, tempo de CPU ou ambos serão desperdiçados. Use *count* com o parâmetro *skip* para copiar uma porção pequena de um arquivo grande ou de um dispositivo.
- *skip=num* — pula os primeiros *num* blocos do arquivo ou do stream de entrada e não os copia na saída.

👍 **Aviso:** *dd* é bastante eficaz, portanto é importante que você saiba o que está fazendo ao executá-lo. É muito fácil corromper arquivos e dados nos dispositivos ao cometer um erro por falta de cuidado. Com frequência, será útil escrever a saída em um arquivo novo se você não estiver certo sobre o que o comando irá fazer.

3.4 Resumo dos nomes dos dispositivos

Às vezes, pode ser difícil encontrar o nome de um dispositivo (por exemplo, ao particionar um disco). A seguir, apresentamos algumas maneiras de descobrir quais são eles:

- Faça uma consulta em *udev* usando *udevadm* (veja a seção 3.5).
- Procure o dispositivo no diretório */sys*.
- Descubra o nome a partir da saída do comando *dmesg* (que exhibe as últimas mensagens do kernel) ou do arquivo de log de sistema do kernel (veja a seção 7.2). Essa saída poderá conter uma descrição dos dispositivos de seu sistema.

- Para um dispositivo de disco que já esteja visível ao sistema, você poderá verificar a saída do comando `mount`.
- Execute `cat /proc/devices` para ver os dispositivos de bloco e de caractere para os quais o seu sistema tenha drivers no momento. Cada linha é composta de um número e de um nome. O número corresponde ao número principal do dispositivo, conforme descrito na seção 3.1. Se puder descobrir qual é o dispositivo a partir do nome, dê uma olhada em `/dev` em busca dos dispositivos de caractere ou de bloco com o número principal correspondente, e você terá encontrado os arquivos do dispositivo.

Entre esses métodos, somente o primeiro é confiável, porém ele exige o `udev`. Se você se vir em uma situação em que `udev` não está disponível, tente os outros métodos, porém tenha em mente que o kernel poderá não ter um arquivo de dispositivo para o seu hardware.

As seções a seguir listam os dispositivos Linux mais comuns e suas convenções de nomenclatura.

3.4.1 Discos rígidos: `/dev/sd*`

A maioria dos discos rígidos conectados a sistemas Linux atuais corresponde a nomes de dispositivo com um prefixo `sd`, por exemplo, `/dev/sda`, `/dev/sdb` e assim por diante. Esses dispositivos representam discos inteiros; o kernel cria arquivos separados de dispositivo, como `/dev/sda1` e `/dev/sda2`, para as partições em um disco.

A convenção de nomenclatura exige algumas explicações. A parte correspondente a `sd` no nome quer dizer *SCSI disk* (disco SCSI). O *SCSI (Small Computer System Interface)* originalmente foi desenvolvido como um padrão de hardware e de protocolo para comunicações entre dispositivos como discos e outros periféricos. Embora o hardware SCSI tradicional não seja usado na maioria dos computadores modernos, o protocolo SCSI está em toda parte graças à sua capacidade de adaptação. Por exemplo, os dispositivos de armazenamento USB o utilizam para se comunicar. A história dos discos SATA é um pouco mais complicada, porém o kernel do Linux ainda usa comandos SCSI em um determinado ponto ao conversar com eles.

Para listar os dispositivos SCSI em seu sistema, utilize um utilitário que percorre os paths dos dispositivos fornecidos pelo `sysfs`. Uma das ferramentas mais sucintas é o `ls SCSI`. Eis o que você pode esperar ao executá-lo:

```
$ ls SCSI
[0:0:0:0] ① disk ② ATA    WDC WD3200AAJS-2 01.0 /dev/sda ③
[1:0:0:0]  cd/dvd Slimtype DVD A  DS8A5SH  XA15 /dev/sr0
```

```
[2:0:0:0] disk FLASH Drive UT_USB20 0.00 /dev/sdb
```

A primeira coluna ❶ identifica o endereço do dispositivo no sistema, a segunda ❷ descreve o tipo do dispositivo e a última ❸ indica o local em que está o arquivo do dispositivo. Todas as demais informações são dos fabricantes.

O Linux atribui dispositivos a arquivos de dispositivo na ordem em que seus drivers encontrarem os dispositivos. Portanto, no exemplo anterior, o kernel encontrou o disco primeiro, o drive óptico em segundo lugar e o flash drive por último.

Infelizmente, esse esquema de atribuição de dispositivos tradicionalmente vem causando problemas quando o hardware é reconfigurado. Suponha, por exemplo, que você tenha um sistema com três discos: */dev/sda*, */dev/sdb* e */dev/sdc*. Se */dev/sdb* explodir e for necessário remover o disco para que o computador possa funcionar novamente, o antigo */dev/sdc* se transformará em */dev/sdb*, e */dev/sdc* não existirá mais. Se você estiver referenciando diretamente os nomes dos dispositivos no arquivo *fstab* (veja a seção 4.2.8), será preciso fazer alterações nesse arquivo para que a situação (ou a maior parte dela) volte ao normal. Para resolver esse problema, a maioria dos sistemas Linux modernos usa o UUID (Universally Unique Identifier, ou Identificador único global – veja a seção 4.2.4) para acessos persistentes a dispositivos de disco.

Essa discussão mal tocou a superfície sobre como usar discos e outros dispositivos de armazenamento em sistemas Linux. Veja o capítulo 4 para obter mais informações sobre o uso de discos. Mais adiante neste capítulo, analisaremos como o suporte para SCSI funciona no kernel do Linux.

3.4.2 Drives de CD e de DVD: */dev/sr**

O Linux reconhece a maioria dos drives de armazenamento ópticos como dispositivos SCSI */dev/sr0*, */dev/sr1* e assim por diante. Entretanto, se o drive utilizar uma interface antiga, ele poderá ser mostrado como um dispositivo PATA, conforme discutiremos posteriormente. Os dispositivos */dev/sr** são somente para leitura e são usados apenas para ler os discos. Para as funcionalidades de escrita e de reescrita de dispositivos ópticos, use os dispositivos SCSI “genéricos” como */dev/sg0*.

3.4.3 Discos rígidos PATA: */dev/hd**

Os dispositivos de bloco do Linux – */dev/hda*, */dev/hdb*, */dev/hdc* e */dev/hdd* – são comuns em versões mais antigas do kernel do Linux e com hardwares mais antigos. São atribuições feitas de modo fixo, de acordo com os dispositivos master (mestre) e slave (escravo) nas interfaces 0 e 1. Ocasionalmente, você poderá encontrar um dispositivo SATA reconhecido como um desses discos. Isso significa que o drive SATA está

executando em modo de compatibilidade, o que atrapalha o desempenho. Verifique suas configurações de BIOS para ver se é possível mudar o controlador SATA para o seu modo nativo.

3.4.4 Terminais: `/dev/tty*`, `/dev/pts/*` e `/dev/tty`

Os *terminais* são dispositivos para transferir caracteres entre um processo de usuário e um dispositivo de I/O, geralmente para saída de texto em uma tela de terminal. A interface do dispositivo de terminal tem uma longa história e remete à época em que os terminais eram dispositivos baseados em máquinas de escrever.

Os dispositivos *pseudoterminais* são terminais emulados que entendem os recursos de I/O de terminais de verdade. Porém, em vez de conversar com um hardware de verdade, o kernel apresenta a interface de I/O a um software, por exemplo, a janela de terminal do shell, em que provavelmente você digitará a maioria de seus comandos.

Dois dispositivos comuns de terminal são `/dev/tty1` (o primeiro console virtual) e `/dev/pts/0` (o primeiro dispositivo de pseudoterminal). O diretório `/dev/pts` em si é um sistema de arquivos dedicado.

O dispositivo `/dev/tty` é o terminal controlador do processo corrente. Se um programa estiver lendo e escrevendo em um terminal no momento, esse dispositivo será sinônimo desse terminal. Um processo não precisa estar conectado a um terminal.

Modos de exibição e consoles virtuais

O Linux tem dois modos principais de exibição: *modo texto* e um *servidor X Window System* (modo gráfico, normalmente por meio de um gerenciador de display). Embora os sistemas Linux, tradicionalmente, sejam inicializados em modo texto, a maioria das distribuições atualmente usa parâmetros de kernel e sistemas temporários de exibição gráfica (bootsplashes como o plymouth) para ocultar totalmente o modo texto enquanto o sistema está sendo inicializado. Em casos como esse, o sistema alterna para o modo gráfico completo próximo ao final do processo de inicialização.

O Linux suporta *consoles virtuais* para multiplexar o display. Cada console virtual pode executar em modo gráfico ou texto. Quando o console virtual estiver em modo texto, você poderá alternar entre os consoles usando a combinação de teclas *Alt-Função* – por exemplo, Alt-F1 levará você para `/dev/tty1`, Alt-F2 para `/dev/tty2` e assim por diante. Muitos deles poderão estar ocupados por um processo `getty` executando um prompt de login, conforme será descrito na seção 7.4.

Um console virtual usado pelo servidor X em modo gráfico é um pouco diferente. Em vez de obter uma atribuição de console virtual a partir da configuração `init`, um servidor

X assume o controle de um console virtual livre, a menos que seja direcionado de modo a utilizar um console virtual específico. Por exemplo, se você tiver processos `getty` executando em `tty1` e em `tty2`, um novo servidor X assumirá o controle de `tty3`. Além disso, depois que o servidor X colocar um console virtual em modo gráfico, normalmente, você deverá pressionar uma combinação de teclas Ctrl-Alt-Função para mudar para outro console virtual, em vez de usar a combinação de teclas Alt-Função mais simples.

O resultado de tudo isso é que, se quiser ver o seu console de texto após o seu sistema ter inicializado, você deverá teclar Ctrl-Alt-F1. Para retornar à sessão X11, tecle Alt-F2, Alt-F3 e assim por diante, até chegar à sessão X.

Se você se deparar com problemas ao alternar entre os consoles devido a um sistema de entrada com mau funcionamento ou outra circunstância, você poderá tentar forçar o sistema a mudar de console por meio do comando `chvt`. Por exemplo, para mudar para `tty1`, execute o comando a seguir como root:

```
# chvt 1
```

3.4.5 Portas seriais: `/dev/ttyS*`

Portas seriais antigas do tipo RS-232 e portas semelhantes são dispositivos de terminal especiais. Não é possível realizar muitas tarefas na linha de comando com dispositivos de portas seriais porque há configurações demais com que se preocupar, por exemplo, baud rate e controle de fluxo.

A porta conhecida como COM1 no Windows é `/dev/ttyS0`; COM2 é `/dev/ttyS1` e assim por diante. Adaptadores de serial plug-in USB aparecem com *USB* e *ACM*, e os nomes `/dev/ttyUSB0`, `/dev/ttyACM0`, `/dev/ttyUSB1`, `/dev/ttyACM1` e assim por diante são usados.

3.4.6 Portas paralelas: `/dev/lp0` e `/dev/lp1`

Representando um tipo de interface que foi amplamente substituído pelo USB, os dispositivos de porta paralela unidirecionais `/dev/lp0` e `/dev/lp1` correspondem a LPT1: e a LPT2: no Windows. Você pode enviar arquivos (por exemplo, um arquivo para impressão) diretamente a uma porta paralela usando o comando `cat`, porém poderá ser necessário enviar um form feed (caractere de alimentação de formulário) ou um reset extra depois. Um servidor de impressão como o CUPS é muito melhor para lidar com interações com uma impressora.

As portas paralelas bidirecionais são `/dev/parport0` e `/dev/parport1`.

3.4.7 Dispositivos de áudio: `/dev/snd/*`, `/dev/dsp`, `/dev/audio` e outros

O Linux tem dois conjuntos de dispositivos de áudio. Há dispositivos separados para a interface de sistema *ALSA* (Advanced Linux Sound Architecture) e o *OSS* (Open Sound System), mais antigo. Os dispositivos ALSA estão no diretório `/dev/snd`, porém é difícil trabalhar com eles diretamente. Os sistemas Linux que usam ALSA suportam dispositivos OSS com compatibilidade para trás se o suporte do kernel para OSS estiver carregado no momento.

Algumas operações rudimentares são possíveis com os dispositivos OSS *dsp* e *audio*. Por exemplo, o computador reproduzirá qualquer arquivo WAV que for enviado para `/dev/dsp`. Entretanto o hardware poderá não fazer o que você espera em virtude da incompatibilidade de frequências. Além do mais, na maioria dos sistemas, o dispositivo geralmente estará ocupado assim que você fizer login.

👍 **Observação:** o áudio no Linux é um assunto confuso por causa das diversas camadas envolvidas. Acabamos de discutir os dispositivos no nível de kernel, porém, normalmente, há servidores no espaço de usuário, como o *pulseaudio*, que administram áudios de diferentes origens e atuam como intermediários entre os dispositivos de áudio e outros processos do espaço de usuário.

3.4.8 Criando arquivos de dispositivo

Em sistemas Linux modernos, você não cria seus próprios arquivos de dispositivo; isso é feito com *devtmpfs* e *udev* (veja a seção 3.5). No entanto é instrutivo saber como isso era feito no passado e, em raras ocasiões, poderá ser necessário criar um pipe nomeado (named pipe).

O comando `mknod` cria um dispositivo. Você deve saber o nome do dispositivo bem como seus números principal (major) e secundário (minor). Por exemplo, criar `/dev/sda1` é uma questão de usar o comando a seguir:

```
# mknod /dev/sda1 b 8 2
```

`b 8 2` especifica um dispositivo de bloco, com um número principal igual a 8 e um número secundário igual a 2. Para dispositivos de caractere ou pipes nomeados, use `c` ou `p` no lugar de `b` (omite os números principal e secundário para pipes nomeados).

Conforme mencionado anteriormente, o comando `mknod` é útil somente para a criação de um pipe nomeado ocasional. Houve uma época em que, às vezes, ele também era útil para criar dispositivos ausentes em modo monousuário durante uma recuperação de sistema.

Em versões mais antigas de Unix e de Linux, manter o diretório `/dev` era um desafio. A

cada upgrade significativo do kernel ou adição de driver, o kernel passava a suportar mais tipos de dispositivo, o que significava que haveria um novo conjunto de números principal e secundário a serem atribuídos aos nomes de arquivos de dispositivo. Manter isso era difícil, de modo que cada sistema tinha um programa MAKEDEV em */dev* para criar grupos de dispositivos. Ao fazer a atualização de seu sistema, era preciso encontrar um upgrade para MAKEDEV e, em seguida, executá-lo para criar novos dispositivos.

Esse sistema estático tornou-se inapropriado, fazendo com que uma substituição se tornasse necessária. A primeira tentativa de corrigir essa situação foi com devfs – uma implementação do espaço de kernel de */dev* que continha todos os dispositivos suportados pelo kernel corrente. Entretanto havia uma série de limitações que levaram ao desenvolvimento do udev e de devtmpfs.

3.5 udev

Já falamos sobre como uma complexidade desnecessária no kernel é perigosa, pois você poderá introduzir instabilidade no sistema com muita facilidade. O gerenciamento de arquivos de dispositivo é um exemplo: você pode criar arquivos de dispositivo no espaço de usuário, então por que você faria isso no kernel? O kernel do Linux pode enviar notificações a um processo no espaço de usuário (chamado *udev*) quando um novo dispositivo for detectado no sistema (por exemplo, quando alguém conectar um pen drive USB). O processo do espaço de usuário na outra extremidade examinará as características do novo dispositivo, criará um arquivo de dispositivo e então realizará qualquer inicialização do dispositivo.

Essa foi a teoria. Infelizmente, na prática, há um problema com essa abordagem – os arquivos de dispositivo são necessários bem cedo no procedimento de inicialização, portanto *udev* deve ser iniciado com antecedência. Para criar arquivos de dispositivo, *udev* não pode depender de nenhum dispositivo que ele deva criar, e deverá realizar o seu startup inicial muito rapidamente, de modo que o restante do sistema não fique travado, esperando *udev* iniciar.

3.5.1 devtmpfs

O sistema de arquivo *devtmpfs* foi desenvolvido em resposta ao problema de disponibilidade de dispositivos durante a inicialização (veja a seção 4.2 para obter mais detalhes sobre sistemas de arquivo). Esse sistema de arquivos é semelhante ao suporte oferecido pelo *devfs* mais antigo, porém foi simplificado. O kernel cria

arquivos de dispositivo conforme forem necessários, porém notifica também `udev` de que um novo dispositivo está disponível. Ao receber esse sinal, `udev` não cria os arquivos de dispositivo, porém realiza a inicialização do dispositivo e processa a notificação. Além do mais, ele cria vários links simbólicos em `/dev` para identificar posteriormente os dispositivos. Você pode encontrar exemplos no diretório `/dev/disk/by-id`, em que cada disco conectado tem uma ou mais entradas.

Por exemplo, considere este disco típico:

```
lrwxrwxrwx 1 root root 9 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671 -> ../../sda
lrwxrwxrwx 1 root root 10 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671-part1 ->
../../sda1
lrwxrwxrwx 1 root root 10 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671-part2 ->
../../sda2
lrwxrwxrwx 1 root root 10 Jul 26 10:23 scsi-SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671-part5 ->
../../sda5
```

O `udev` nomeia os links por tipo de interface e, em seguida, por informações de fabricante e modelo, número de série e partição (se for aplicável).

Mas como o `udev` sabe quais links simbólicos deve criar e como ele os cria? A próxima seção descreve como ele faz o seu trabalho. Entretanto não é preciso saber isso para continuar lendo este livro. Com efeito, se essa for a primeira vez que você está dando uma olhada nos dispositivos Linux, é melhor avançar para o próximo capítulo e começar a aprender a usar os discos.

3.5.2 Operação e configuração do `udev`

O daemon `udev` funciona da seguinte maneira:

1. O kernel envia um evento de notificação chamado *uevent* ao `udev` por meio de um link de rede interno.
2. O `udev` carrega todos os atributos de *uevent*.
3. O `udev` faz parse de suas regras e realiza ações ou define mais atributos, de acordo com essas regras.

Um *uevent* de entrada recebido do kernel pelo `udev` pode ter uma aparência semelhante a:

```
ACTION=change
DEVNAME=sde
DEVPATH=/devices/pci0000:00/0000:00:1a.0/usb1/1-1/1-1.2/1-1.2:1.0/host4/target4:0:0/4:0:0:3/block/sde
DEVTYPE=disk
DISK_MEDIA_CHANGE=1
MAJOR=8
```



```
ID_REVISION=01.03E10
ID_SERIAL=WDC_WD3200AAJS-22L7A0_WD-WMAV2FU80671
--trecho omitido--
```

A importação agora define o ambiente de modo que todos os nomes de variáveis nessa saída sejam definidos com os valores mostrados. Por exemplo, qualquer regra que se seguir a partir de agora irá reconhecer `ENV{ID_TYPE}` como disco.

Em particular, é interessante observar `ID_SERIAL`. Em cada uma das regras, esta condicional aparece em segundo lugar:

```
ENV{ID_SERIAL}!="?*"
```

Isso significa que `ID_SERIAL` será verdadeiro somente se não estiver definido. Sendo assim, se *estiver* definido, a condicional será falsa, toda a regra corrente será falsa e `udev` prosseguirá para a próxima regra.

Então qual é o propósito? O objeto dessas duas regras (e de várias outras em torno delas no arquivo) é encontrar o número de série do dispositivo de disco. Com `ENV{ID_SERIAL}` definido, `udev` poderá agora avaliar esta regra:

```
KERNEL=="sd*|sr*|cciss*", ENV{DEVTYPE}=="disk", ENV{ID_SERIAL}!="?*",
SYMLINK+="disk/by-id/${env{ID_BUS}}-${env{ID_SERIAL}}"
```

Você pode ver que essa regra exige que `ENV{ID_SERIAL}` esteja definido, e ela tem uma diretiva:

```
SYMLINK+="disk/by-id/${env{ID_BUS}}-${env{ID_SERIAL}}"
```

Ao encontrar essa diretiva, `udev` adicionará um link simbólico para o dispositivo que está chegando. Então agora sabemos de onde os links simbólicos vieram!

Você pode estar se perguntando como diferenciar uma expressão condicional de uma diretiva. As condicionais são indicadas por dois sinais de igualdade (`==`) ou um ponto de exclamação (`!=`); as diretivas, por um único sinal de igualdade (`=`), um sinal de adição (`+=`) ou dois-pontos e sinal de igualdade (`=:`).

3.5.3 udevadm

O programa `udevadm` é uma ferramenta de administração para `udev`. Você pode recarregar as regras de `udev` e disparar eventos, porém os recursos mais eficazes de `udevadm` talvez sejam as capacidades de procurar e explorar dispositivos de sistema e monitorar `uevents` à medida que `udev` os receber do kernel. A única parte complicada é que a sintaxe do comando pode se tornar um pouco complexa.

Vamos começar analisando um dispositivo do sistema. Retomando o exemplo da seção 3.5.2, para ver todos os atributos de `udev` utilizados e gerados em conjunto com as

regras para um dispositivo como `/dev/sda`, execute o comando a seguir:

```
$ udevadm info --query=all --name=/dev/sda
```

A saída terá a seguinte aparência:

```
P: /devices/pci0000:00/0000:00:1f.2/host0/target0:0:0:0:0:0/block/sda
N: sda
S: disk/by-id/ata-WDC_WD3200AAJS-22L7A0_WD-WMAV2FU80671
S: disk/by-id/scsi-SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671
S: disk/by-id/wwn-0x50014ee057faef84
S: disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0
E: DEVLINKS=/dev/disk/by-id/ata-WDC_WD3200AAJS-22L7A0_WD-WMAV2FU80671 /dev/disk/by-id/scsi-SATA_WDC_WD3200AAJS-_WD-WMAV2FU80671 /dev/disk/by-id/wwn-0x50014ee057faef84 /dev/disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0
E: DEVNAME=/dev/sda
E: DEVPATH=/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0:0:0:0/block/sda
E: DEVTYPE=disk
E: ID_ATA=1
E: ID_ATA_DOWNLOAD_MICROCODE=1
E: ID_ATA_FEATURE_SET_AAM=1
--trecho omitido--
```

O prefixo em cada linha indica um atributo ou outra característica do dispositivo. Nesse caso, P: na parte superior é o path sysfs do dispositivo, N: é o nó do dispositivo (ou seja, o nome dado ao arquivo `/dev`), S: indica um link simbólico para o nó do dispositivo que `udev` colocou em `/dev` de acordo com suas regras e E: corresponde a informações adicionais do dispositivo, extraídas das regras de `udev`. (Havia muito mais informações de saída nesse exemplo do que era necessário mostrar aqui; procure executar o comando por conta própria para ter uma noção do que ele faz.)

3.5.4 Monitorando dispositivos

Para monitor uevents com `udevadm`, utilize o comando `monitor`:

```
$ udevadm monitor
```

A saída (por exemplo, ao inserir um dispositivo de memória flash) se parecerá com este exemplo resumido:

```
KERNEL[658299.569485] add /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2 (usb)
KERNEL[658299.569667] add /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2:1.0 (usb)
KERNEL[658299.570614] add /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2:1.0/host15 (scsi)
KERNEL[658299.570645] add /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2:1.0/host15/scsi_host/host15 (scsi_host)
UDEV [658299.622579] add /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2 (usb)
UDEV [658299.623014] add /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2:1.0 (usb)
```

```
UDEV [658299.623673] add /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2:1.0/host15 (scsi)
UDEV [658299.623690] add /devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2:1.0/host15/
scsi_host/host15 (scsi_host)
--trecho omitido--
```

Há duas cópias de cada mensagem nessa saída porque o comportamento-padrão consiste em exibir tanto a mensagem de entrada do kernel (marcada com KERNEL) quanto a mensagem que `udev` envia para outros programas quando ele acaba de processar e de filtrar o evento. Para ver somente os eventos do kernel, adicione a opção `--kernel`, e para ver somente os eventos de saída, use `--udev`. Para ver todo o `uevent` de entrada, incluindo os atributos conforme mostrados na seção 3.5.2, utilize a opção `--property`.

Você também pode filtrar eventos de acordo com o subsistema. Por exemplo, para ver somente as mensagens de kernel que dizem respeito a mudanças no subsistema SCSI, utilize o comando a seguir:

```
$ udevadm monitor --kernel --subsystem-match=scsi
```

Para mais informações sobre `udevadm`, consulte a página de manual `udevadm(8)`.

Há muito mais a dizer sobre o `udev`. Por exemplo, o sistema D-Bus para comunicação entre processos tem um daemon chamado `udisks-daemon` que fica ouvindo os eventos de saída de `udev` para conectar discos automaticamente e notificar outros softwares desktop de que um novo disco encontra-se agora disponível.

3.6 Em detalhes: SCSI e o kernel do Linux

Nesta seção, daremos uma olhada no suporte a SCSI do kernel do Linux como uma maneira de explorar parte da arquitetura desse kernel. Não é preciso conhecer nenhuma dessas informações para usar discos, portanto, se estiver com pressa de usar um, vá para o capítulo 4. Além do mais, o material aqui é mais avançado e teórico por natureza em relação ao que vimos até agora, portanto, se você quiser continuar com a mão na massa, definitivamente, pule para o próximo capítulo.

Vamos começar com um pouco de história. A configuração do hardware SCSI tradicional consiste de um adaptador de host ligado a uma cadeia de dispositivos por meio de um barramento SCSI, conforme mostrado na figura 3.1. O adaptador de host é conectado a um computador. O adaptador de host e os dispositivos têm um SCSI ID, e pode haver 8 ou 16 IDs por barramento, de acordo com a versão de SCSI. Você poderá ouvir o termo *alvo SCSI* (SCSI target) para se referir a um dispositivo e o seu SCSI ID.

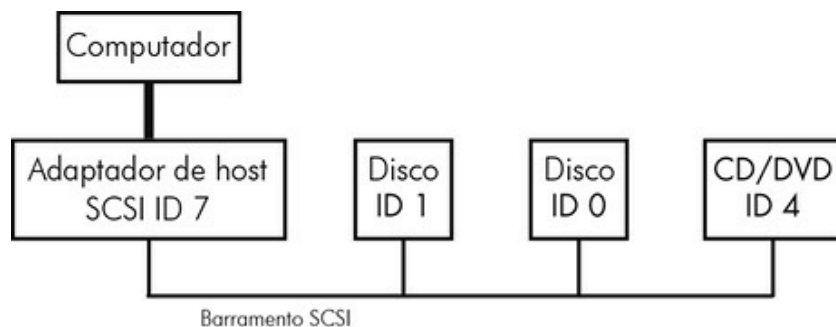


Figura 3.1 – Barramento SCSI com adaptador de host e dispositivos.

O adaptador de host se comunica com os dispositivos por meio do conjunto de comandos SCSI em uma relação ponto a ponto (peer-to-peer); os dispositivos enviam respostas de volta ao adaptador de host. O computador não está diretamente conectado à cadeia de dispositivos, portanto ele deve passar pelo adaptador de host para se comunicar com os discos e com outros dispositivos. Normalmente, o computador envia comandos SCSI para o adaptador de host para acessar os dispositivos, e os dispositivos enviam respostas de volta por meio do adaptador de host.

Versões mais novas de SCSI, como o *SAS* (Serial Attached SCSI), oferecem um desempenho excepcional, mas é provável que você não vá encontrar verdadeiros dispositivos SCSI na maioria dos computadores. Com mais frequência, você irá encontrar dispositivos de armazenamento USB que usam comandos SCSI. Além do mais, os dispositivos que suportam ATAPI (por exemplo, drives de CD/DVD-ROM) usam uma versão do conjunto de comandos SCSI.

Os discos SATA também aparecem em seu sistema como dispositivos SCSI por causa de uma camada de tradução em libata (veja a seção 3.6.2). Alguns controladores SATA (especialmente os controladores RAID de alto desempenho) realizam essa tradução no hardware.

Como tudo isso se encaixa? Considere os dispositivos mostrados no sistema a seguir:

```

$ ls SCSI
[0:0:0:0] disk ATA WDC WD3200AAJS-2 01.0 /dev/sda
[1:0:0:0] cd/dvd Slimtype DVD A DS8A5SH XA15 /dev/sr0
[2:0:0:0] disk USB2.0 CardReader CF 0100 /dev/sdb
[2:0:0:1] disk USB2.0 CardReader SM XD 0100 /dev/sdc
[2:0:0:2] disk USB2.0 CardReader MS 0100 /dev/sdd
[2:0:0:3] disk USB2.0 CardReader SD 0100 /dev/sde
[3:0:0:0] disk FLASH Drive UT_USB20 0.00 /dev/sdf
  
```

Os números entre colchetes são, da esquerda para a direita, o número do adaptador de host SCSI, o número do barramento SCSI, o SCSI ID do dispositivo e o LUN (Logical Unit Number, ou Número de unidade lógica – outra subdivisão de um dispositivo).

Nesse exemplo, há quatro adaptadores conectados (`scsi0`, `scsi1`, `scsi2` e `scsi3`), cada qual com um único barramento (todos com número de barramento 0) e somente um dispositivo em cada barramento (todos com alvo 0). Porém o leitor de cartão USB em 2:0:0 tem quatro unidades lógicas – uma para cada tipo de cartão de memória flash que pode ser inserido. O kernel atribuiu um arquivo diferente de dispositivo para cada unidade lógica.

A figura 3.2 mostra a hierarquia de drivers e de interfaces no kernel para essa configuração de sistema em particular, desde os device drivers individuais até os drivers de bloco. A figura não inclui os drivers SCSI genéricos (`sg`).

Embora essa seja uma estrutura grande e possa parecer intimidadora à primeira vista, o fluxo de dados na figura é bem linear. Vamos começar a detalhá-la observando o subsistema SCSI e suas três camadas de drivers:

- A camada superior cuida de operações para uma classe de dispositivo. Por exemplo, o driver `sd` (SCSI disk) está nessa camada; ele sabe como traduzir solicitações da interface de dispositivo de blocos do kernel para comandos específicos de disco no protocolo SCSI e vice-versa.
- A camada intermediária atua como moderadora e encaminha as mensagens SCSI entre as camadas superior e inferior, além de monitorar todos os barramentos SCSI e os dispositivos conectados ao sistema.
- A camada inferior cuida de ações específicas do hardware. Os drivers nesse local enviam mensagens de saída do protocolo SCSI para adaptadores de host ou hardwares específicos e extraem mensagens de entrada do hardware. O motivo para essa separação da camada superior está no fato de que, embora as mensagens SCSI sejam uniformes para uma classe de dispositivo (por exemplo, a classe disco), diferentes tipos de adaptadores de host têm procedimentos variados para enviar as mesmas mensagens.

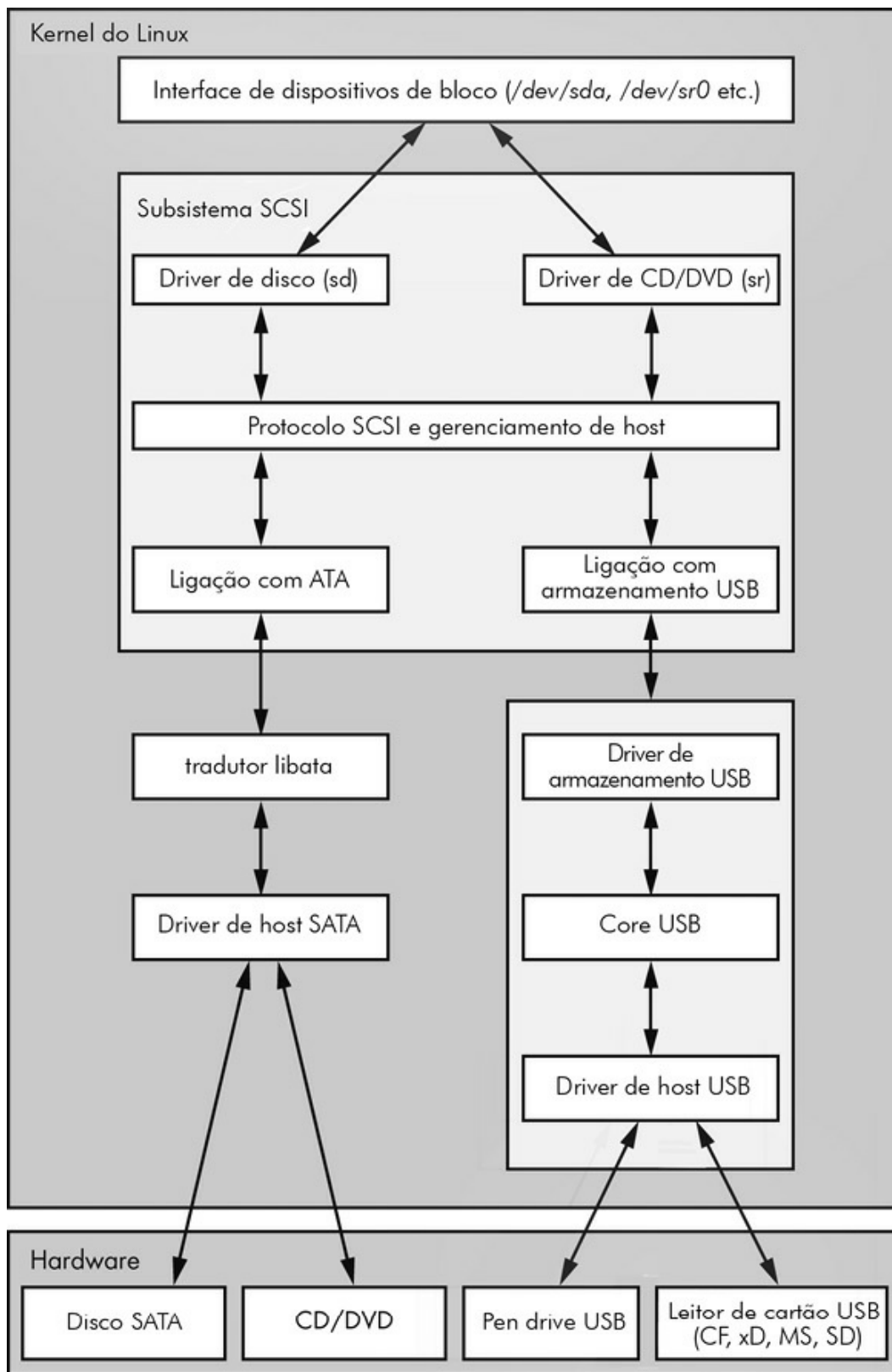


Figura 3.2 – Esquema do subsistema SCSI do Linux.

As camadas superior e inferior contêm vários drivers diferentes, porém é importante lembrar que, para qualquer arquivo de dispositivo em particular de seu sistema, o kernel usa um driver da camada superior e um driver da camada inferior. Para o disco

em `/dev/sda` em nosso exemplo, o kernel utiliza o driver `sd` da camada superior e o driver de ligação com ATA da camada inferior.

Há ocasiões em que você poderá usar mais de um driver de camada superior para um dispositivo de hardware (veja a seção 3.6.3). Para verdadeiros dispositivos de hardware SCSI, por exemplo, um disco conectado a um adaptador de host SCSI ou um controlador RAID de hardware, os drivers da camada inferior conversam diretamente com o hardware que está abaixo. Entretanto, para a maioria dos hardwares que você encontrar conectados ao subsistema SCSI, a história será diferente.

3.6.1 Armazenamento USB e SCSI

Para o subsistema SCSI conversar com um hardware comum de armazenamento USB, como mostrado na figura 3.2, o kernel precisa de mais do que apenas um driver SCSI de camada inferior. O pen drive USB representado por `/dev/sdf` entende comandos SCSI, porém, para realmente se comunicar com o drive, o kernel deverá saber como conversar por meio do sistema USB.

De modo abstrato, o USB é bem semelhante ao SCSI – ele tem classes de dispositivo, barramentos e controladores de host. Desse modo, não é de surpreender que o kernel do Linux inclua um subsistema USB de três camadas que lembra muito o subsistema SCSI, com drivers para classes de dispositivo na parte superior, um núcleo para gerenciamento de barramento no meio e drivers para controladores de hosts na parte inferior. Assim como um subsistema SCSI passa comandos SCSI entre seus componentes, o subsistema USB passa mensagens USB entre seus componentes. Há até mesmo um comando `lsusb`, semelhante a `lscsi`.

A parte em que realmente estamos interessados, nesse caso, é o driver de armazenamento USB na parte superior. Esse driver atua como um tradutor. De um lado, o driver fala SCSI e de outro, fala USB. Como o hardware de armazenamento inclui comandos SCSI em suas mensagens USB, o driver tem um trabalho relativamente fácil: em sua maior parte, ele reempacota os dados.

Com os subsistemas SCSI e USB instalados, você terá quase tudo que é necessário para conversar com o pen drive. O último elo faltando é o driver de camada inferior do subsistema SCSI, pois o driver de armazenamento USB faz parte do subsistema USB, e não do subsistema SCSI. (Por razões de organização, os dois subsistemas não devem compartilhar um driver.) Para fazer os subsistemas conversarem um com o outro, um driver de ligação SCSI simples na camada inferior conecta-se com o driver de armazenamento do subsistema USB.

3.6.2 SCSI e ATA

O disco rígido SATA e o drive óptico mostrados na figura 3.2 utilizam a mesma interface SATA. Para conectar os drivers do kernel específicos do SATA ao subsistema SCSI, o kernel emprega um driver de ligação, como ocorre com os drives USB, porém com um sistema diferente e com complicações adicionais. O drive óptico fala ATAPI, que é uma versão dos comandos SCSI codificada no protocolo ATA. Entretanto o disco rígido não usa ATAPI e não codifica nenhum comando SCSI!

O kernel do Linux utiliza parte de uma biblioteca chamada libata para reconciliar os drives SATA (e ATA) com o subsistema SCSI. Para os drives ópticos que falam ATAPI, essa é uma tarefa relativamente simples, que consiste em empacotar e extrair comandos SCSI do protocolo ATA. Porém, para o disco rígido, a tarefa é muito mais complicada, pois a biblioteca deve executar uma tradução completa do comando.

A tarefa do drive óptico é semelhante a digitar um livro em inglês em um computador. Não é necessário entender sobre o que é o livro para fazer esse trabalho, e não é preciso nem mesmo saber inglês. Porém a tarefa para o disco rígido é mais semelhante a ler um livro em alemão e digitá-lo no computador como uma tradução para o inglês. Nesse caso, é preciso entender os dois idiomas, além do conteúdo do livro.

Apesar dessa dificuldade, o libata realiza essa tarefa e possibilita conectar o subsistema SCSI a interfaces e dispositivos ATA/SATA. (Normalmente, há mais drivers envolvidos além do único driver de host SATA mostrado na figura 3.2, porém eles não estão sendo apresentados por questão de simplicidade.)

3.6.3 Dispositivos SCSI genéricos

Quando um processo do espaço de usuário se comunica com o subsistema SCSI, normalmente ele faz isso por meio da camada de dispositivos de bloco e/ou outro serviço do kernel que esteja acima de um driver de classe de dispositivo SCSI (como *sd* ou *sr*). Em outras palavras, a maioria dos processos de usuário jamais precisará saber alguma coisa sobre dispositivos SCSI ou seus comandos.

No entanto os processos de usuário podem ignorar os drivers de classes de dispositivo e executar comandos do protocolo SCSI diretamente nos dispositivos por meio de *dispositivos genéricos*. Por exemplo, considere o sistema descrito na seção 3.6, porém, dessa vez, dê uma olhada no que acontece quando a opção *-g* é adicionada a `ls SCSI` para mostrar os dispositivos genéricos:

```
$ ls SCSI -g
```

```
[0:0:0:0] disk ATA WDC WD3200AAJS-2 01.0 /dev/sda ①/dev/sg0
```

```
[1:0:0:0] cd/dvd Slimtype DVD A DS8A5SH XA15 /dev/sr0 /dev/sg1
```



```

[2:0:0:0] disk USB2.0 CardReader CF 0100 /dev/sdb /dev/sg2
[2:0:0:1] disk USB2.0 CardReader SM XD 0100 /dev/sdc /dev/sg3
[2:0:0:2] disk USB2.0 CardReader MS 0100 /dev/sdd /dev/sg4
[2:0:0:3] disk USB2.0 CardReader SD 0100 /dev/sde /dev/sg5
[3:0:0:0] disk FLASH Drive UT_USB20 0.00 /dev/sdf /dev/sg6

```

Além do arquivo usual de dispositivo de bloco, cada entrada lista um arquivo de dispositivo SCSI genérico na última coluna em ❶. Por exemplo, o dispositivo genérico para o drive óptico em */dev/sr0* é */dev/sg1*.

Por que você iria querer usar um dispositivo SCSI genérico? A resposta tem a ver com a complexidade do código no kernel. À medida que as tarefas se tornam mais complicadas, é melhor deixá-las fora do kernel. Considere a escrita e a leitura de CD/DVD. A escrita não só é significativamente mais difícil do que a leitura, como também nenhum serviço crítico do sistema depende da ação de escrita. Um programa do espaço de usuário pode deixar a escrita um pouco mais ineficiente do que faria um serviço do kernel, porém esse programa será muito mais simples de ser criado e mantido do que um serviço do kernel, e os bugs não irão ameaçar o espaço do kernel. Sendo assim, para escrever em um disco óptico no Linux, você deve executar um programa que converse com um dispositivo SCSI genérico, por exemplo, */dev/sg1*. Porém, em virtude da relativa simplicidade da leitura quando comparada à escrita, você continuará lendo do dispositivo usando o driver de dispositivo óptico *sr* especializado do kernel.

3.6.4 Vários métodos de acesso para um único dispositivo

Os dois pontos de acesso (*sr* e *sg*) para um drive óptico a partir do espaço de usuário estão sendo mostrados para o subsistema SCSI do Linux na figura 3.3 (todos os drivers abaixo da camada SCSI inferior foram omitidos). O processo A lê do drive usando o driver *sr* e o processo B escreve no drive com o driver *sg*. No entanto processos como esses dois normalmente não são executados simultaneamente para acessar o mesmo dispositivo.

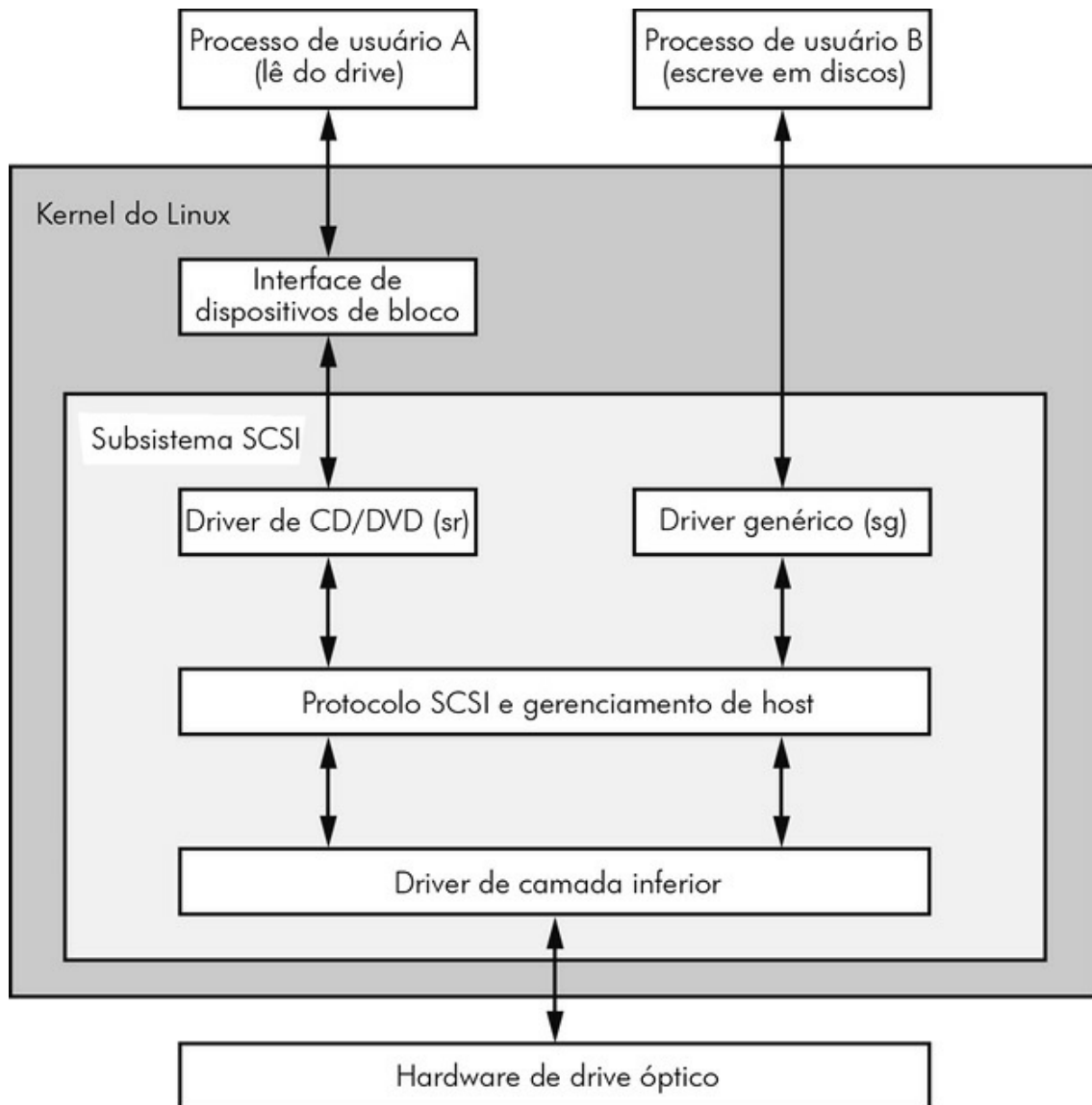


Figura 3.3 – Esquema do driver de dispositivo óptico.

Na figura 3.3, o processo A lê do dispositivo de bloco. Porém os processos de usuário realmente leem dados dessa maneira? Normalmente, a resposta é não – não diretamente. Há outras camadas sobre os dispositivos de bloco e mais pontos de acesso para discos rígidos, como você verá no próximo capítulo.

CAPÍTULO 4

Discos e sistemas de arquivos

No capítulo 3, discutimos alguns dos dispositivos de disco de nível mais alto disponibilizados pelo kernel. Neste capítulo, discutiremos em detalhes como trabalhar com discos em um sistema Linux. Você aprenderá a particionar discos, criar e manter os sistemas de arquivos usados nas partições de disco e trabalhar com áreas de swap.

Lembre-se de que os dispositivos de disco têm nomes como `/dev/sda` – o primeiro disco do subsistema SCSI. Esse tipo de dispositivo de bloco representa o disco todo, porém há vários componentes e camadas diferentes em um disco.

A figura 4.1 mostra o esquema de um disco Linux típico (observe que a figura não está em escala). À medida que avançar neste capítulo, você aprenderá em que lugar cada peça se encaixa.

As *partições* são subdivisões do disco todo. No Linux, elas são representadas com um número depois do dispositivo de bloco completo e, sendo assim, têm nomes de dispositivo como `/dev/sda1` e `/dev/sdb3`. O kernel apresenta cada partição como um dispositivo de bloco, exatamente como faria com um disco completo. As partições estão definidas em uma pequena área do disco chamada de *tabela de partição*.

 **Observação:** várias partições de dados antigamente eram comuns em sistemas com discos grandes, pois PCs mais antigos podiam inicializar somente a partir de determinadas partes do disco. Além disso, os administradores usavam partições para reservar determinada quantidade de espaço para áreas do sistema operacional; por exemplo, eles não queriam que os usuários pudessem ocupar todo o sistema e impedissem o funcionamento de serviços críticos. Essa prática não é exclusiva do Unix; você ainda encontrará muitos sistemas Windows novos com várias partições em um único disco. Além do mais, a maioria dos sistemas tem uma partição separada de swap.

Embora o kernel torne possível acessar tanto um disco inteiro quanto uma de suas partições ao mesmo tempo, normalmente você não fará isso, a menos que esteja copiando o disco todo.

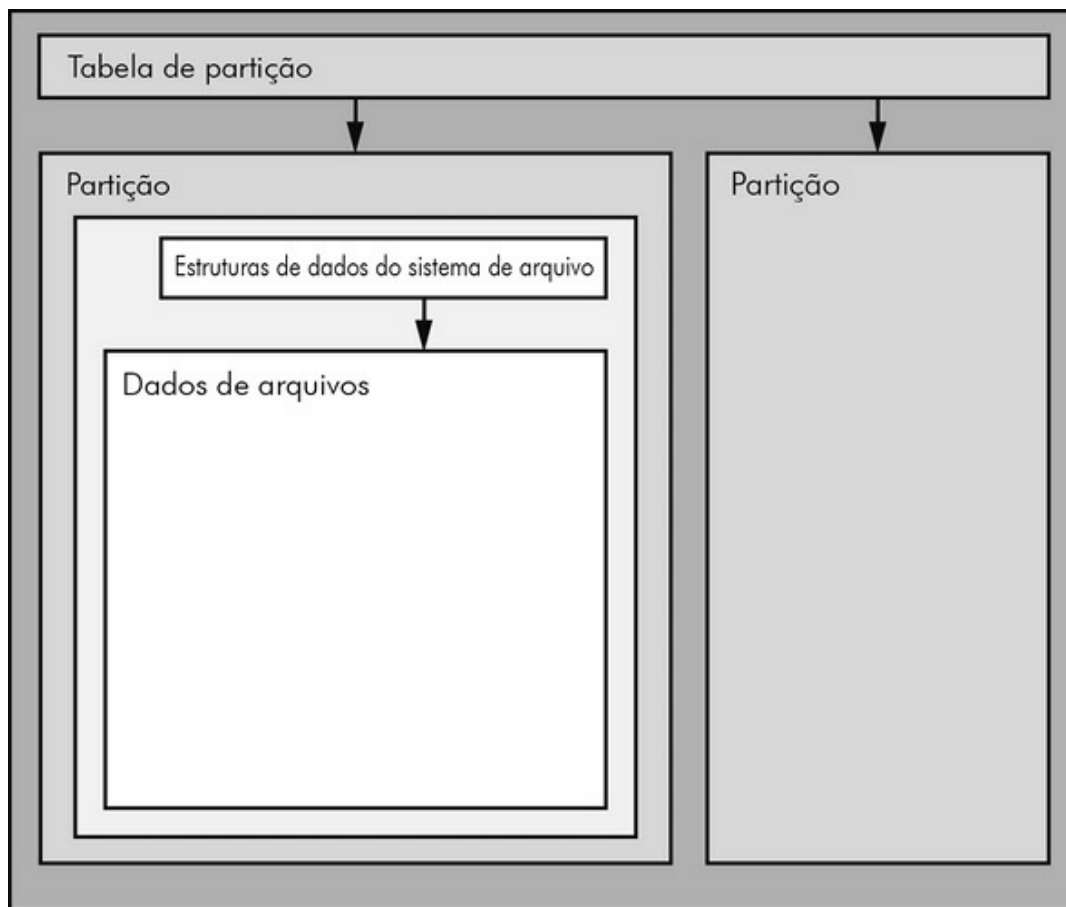


Figura 4.1 – Esquema de um disco Linux típico.

A próxima camada após a partição é o *sistema de arquivos* (filesystem) – o banco de dados de arquivos e diretórios com o qual você está acostumado a interagir no espaço de usuário. Iremos explorar os sistemas de arquivos na seção 4.2.

Como você pode ver na figura 4.1, se você quiser acessar os dados em um arquivo, será necessário obter a localização correta da partição a partir da tabela de partição e, em seguida, pesquisar o banco de dados do sistema de arquivos nessa partição em busca dos dados de arquivo desejados.

Para acessar dados em um disco, o kernel do Linux usa o sistema de camadas mostrado na figura 4.2. O subsistema SCSI e tudo o mais descrito na seção 3.6 estão representados por uma única caixa. (Observe que você pode trabalhar com o disco por meio do sistema de arquivos como também diretamente, por meio dos dispositivos de disco. Você usará ambas as opções neste capítulo.)

Para entender como tudo se encaixa, vamos começar pelas partições na parte inferior.

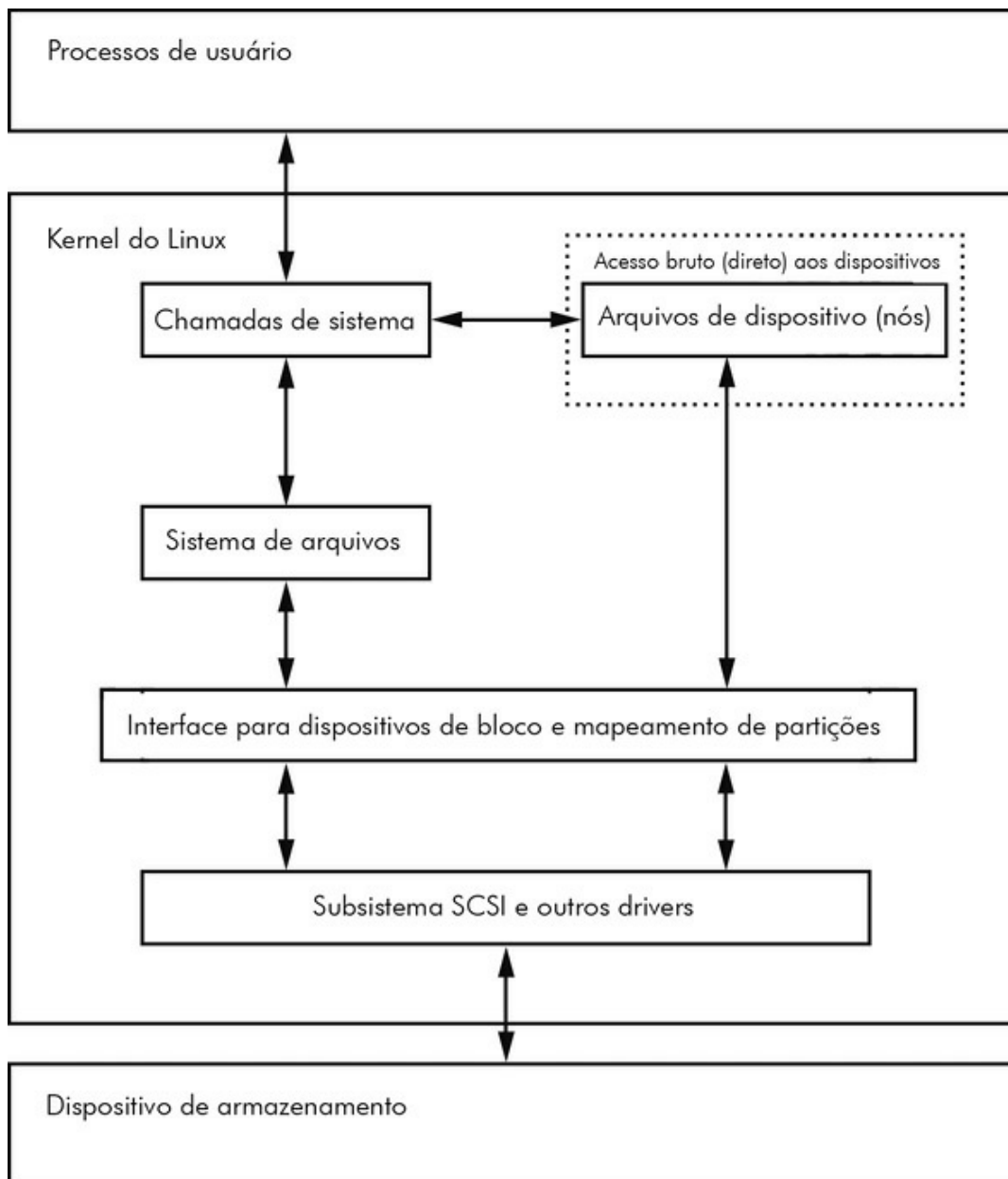


Figura 4.2 – Esquema do kernel para acesso a disco.

4.1 Particionando dispositivos de disco

Há vários tipos de tabelas de partição. A tabela tradicional é aquela que se encontra no *MBR* (Master Boot Record). Um padrão mais novo que começou a ganhar força é o *GPT* (Globally Unique Identifier Partition Table).

A seguir, apresentamos uma visão geral das várias ferramentas de partição disponíveis no Linux:

- *parted* – uma ferramenta baseada em texto que suporta tanto MBR quanto GPT.
- *gparted* – uma versão gráfica de *parted*.

- `fdisk` — é a ferramenta tradicional de particionamento de disco baseada em texto do Linux. O `fdisk` não suporta GPT.
- `gdisk` — é uma versão do `fdisk` que suporta GPT, mas não o MBR.

Pelo fato de suportar tanto MBR quanto GPT, usaremos o `parted` neste livro. Entretanto muitas pessoas preferem a interface do `fdisk`, e não há nada de errado nisso.

 **Observação:** embora `parted` possa criar e redimensionar sistemas de arquivos, não o use para manipulá-los, pois você poderá ficar facilmente confuso. Há uma diferença crucial entre particionamento e manipulação de sistemas de arquivos. A tabela de partição define fronteiras simples no disco, enquanto um sistema de arquivos é um sistema de dados muito mais sofisticado. Por esse motivo, utilizaremos `parted` para particionamento, porém usaremos utilitários diferentes para a criação de sistemas de arquivos (veja a seção 4.2.2). Até mesmo a documentação de `parted` incentiva você a criar sistemas de arquivos separadamente.

4.1.1 Visualizando uma tabela de partição

Você pode visualizar a tabela de partição de seu sistema usando `parted -l`. A seguir, temos um exemplo de saída de dois dispositivos de disco com dois tipos de tabela de partição diferentes:

```
# parted -l
Model: ATA WDC WD3200AAJS-2 (scsi)
Disk /dev/sda: 320GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos

Number Start End Size Type File system Flags
1 1049kB 316GB 316GB primary ext4 boot
2 316GB 320GB 4235MB extended
5 316GB 320GB 4235MB logical linux-swap(v1)

Model: FLASH Drive UT_USB20 (scsi)
Disk /dev/sdf: 4041MB
Sector size (logical/physical): 512B/512B
Partition Table: gpt

Number Start End Size File system Name Flags
1 17.4kB 1000MB 1000MB myfirst
2 1000MB 4040MB 3040MB mysecond
```

O primeiro dispositivo `/dev/sda` usa a tabela de partição MBR tradicional (chamada de “`msdos`” pelo `parted`), e o segundo contém uma tabela GPT. Observe que há diferentes parâmetros para cada tabela de partição, pois as tabelas em si são diferentes. Em particular, não há nenhuma coluna `Name` (Nome) para a tabela MBR porque não há nomes nesse esquema. (Escolhi arbitrariamente os nomes `myfirst` e `mysecond` na tabela GPT.)

A tabela MBR nesse exemplo contém partições primária (primary), estendida (extended) e lógica (logical). Uma *partição primária* é uma subdivisão normal do disco; a partição 1 é desse tipo. O MBR básico tem um limite de quatro partições primárias, portanto, se você quiser mais de quatro, defina uma partição como uma *partição estendida*. A seguir, subdivida a partição estendida em *partições lógicas* que o sistema operacional poderá usar como faria com qualquer outra partição. Nesse exemplo, a partição 2 é uma partição estendida que contém a partição lógica 5.

👍 **Observação:** o sistema de arquivos que parted lista não é necessariamente o campo de ID de sistema definido na maioria das entradas MBR. O ID de sistema do MBR é somente um número; por exemplo, 83 é uma partição Linux e 82 é o swap Linux. Desse modo, parted procura, ele mesmo, determinar um sistema de arquivos. Se você realmente precisar saber o ID de sistema em um MBR, use `fdisk -l`.

Leitura inicial do kernel

Ao ler inicialmente a tabela MBR, o kernel do Linux gera a saída de debugging a seguir (lembre-se de que isso pode ser visto com `dmesg`):

```
sda: sda1 sda2 < sda5 >
```

A saída `sda2 < sda5 >` indica que `/dev/sda2` é uma partição estendida contendo uma partição lógica `/dev/sda5`. Normalmente, você irá ignorar as partições estendidas porque, em geral, irá querer acessar somente as partições lógicas contidas nelas.

4.1.2 Alterando as tabelas de partição

Visualizar tabelas de partição é uma operação relativamente simples e inofensiva. Alterar tabelas de partição também é relativamente fácil, porém há riscos envolvidos ao fazer esse tipo de mudança no disco. Tenha em mente o seguinte:

- Alterar a tabela de partição faz com que seja bem difícil recuperar qualquer dado das partições que você apagar, pois isso altera o ponto inicial de referência de um sistema de arquivos. Garanta que haja um backup se o disco que você estiver particionando contiver dados críticos.
- Certifique-se de que nenhuma partição de seu disco-alvo esteja em uso no momento. Essa é uma preocupação, pois a maioria das distribuições Linux monta automaticamente qualquer sistema de arquivos detectado. (Veja a seção 4.2.3 para mais informações sobre montagem e desmontagem.)

Quando estiver pronto, escolha o seu programa de particionamento. Se quiser usar `parted`, você poderá usar o utilitário `parted` de linha de comando ou uma interface gráfica como `gparted`; para uma interface de estilo `fdisk`, use `gdisk` se estiver usando particionamento GPT. Esses utilitários têm ajuda online e são fáceis de aprender. (Tente

usá-los em um pen drive ou em algo semelhante se não tiver nenhum disco sobrando.)

Apesar do que foi dito, há uma diferença importante na maneira pela qual o `fdisk` e o `parted` funcionam. Com `fdisk`, a sua nova tabela de partição é criada antes de as mudanças propriamente ditas serem feitas no disco; o `fdisk` fará as alterações somente quando você sair do programa. No entanto, com `parted`, as partições são criadas, modificadas e removidas *à medida que os comandos forem executados*. Você não terá a oportunidade de revisar a tabela de partição antes de alterá-la.

Essas diferenças também são importantes para entender como esses dois utilitários interagem com o kernel. Tanto `fdisk` quanto `parted` modificam as partições totalmente no espaço de usuário; não há necessidade de disponibilizar suporte do kernel para reescrever uma tabela de partição porque o espaço de usuário pode ler e modificar todo um dispositivo de bloco.

Em algum momento, porém, o kernel deverá ler a tabela de partição para apresentar as partições como dispositivos de bloco. O utilitário `fdisk` utiliza um método relativamente simples: após modificar a tabela de partição, `fdisk` realiza uma única chamada de sistema no disco para dizer ao kernel que ele deve reler a tabela de partição. O kernel então gera uma saída de debugging que poderá ser visualizada com `dmesg`. Por exemplo, se você criar duas partições em `/dev/sdf`, você verá isto:

```
sdf: sdf1 sdf2
```

Em comparação, as ferramentas `parted` não usam essa chamada de sistema válida para todo o disco. Em vez disso, elas sinalizam o kernel quando partições individuais são alteradas. Após o processamento de uma única mudança de partição, o kernel não irá gerar a saída de debugging anterior.

Há algumas maneiras de ver as alterações de partição:

- Use `udevadm` para observar as mudanças de eventos do kernel. Por exemplo, `udevadm monitor --kernel` mostrará os dispositivos de partição antigos sendo removidos e os novos sendo adicionados.
- Verifique `/proc/partitions` para obter informações completas de partição.
- Verifique `/sys/block/device/` para ver as interfaces de sistema de partições alteradas ou `/dev` para ver os dispositivos de partição alterados.

Se realmente for preciso ter certeza de que uma tabela de partição foi modificada, você poderá realizar uma chamada de sistema de estilo antigo usada pelo `fdisk` usando o comando `blockdev`. Por exemplo, para forçar o kernel a recarregar a tabela de partição em `/dev/sdf`, execute este comando:


```
# blockdev --rereadpt /dev/sdf
```

Nesse ponto, você sabe tudo o que é necessário sobre particionamento de discos. No entanto, se estiver interessado em conhecer mais alguns detalhes sobre discos, continue lendo. Do contrário, pule para a seção 4.2 para saber como colocar um sistema de arquivos no disco.

4.1.3 Geometria do disco e da partição

Qualquer dispositivo com partes móveis introduz complexidade em um sistema de software, pois há elementos físicos que resistem à abstração. Um disco rígido não é nenhuma exceção; mesmo que você possa pensar em um disco rígido como um dispositivo de bloco com acesso aleatório a qualquer bloco, há sérias consequências no desempenho se você não tomar cuidado em relação ao modo como os dados são dispostos no disco. Considere as propriedades físicas do disco de um só prato mostrado na figura 4.3.

O disco é constituído de um prato giratório em um eixo, com um cabeçote conectado a um braço móvel que pode varrer o raio do disco. À medida que o disco gira sob o cabeçote, esse lê os dados. Quando o braço estiver em uma posição, o cabeçote poderá ler dados somente de um círculo fixo. Esse círculo chama-se *cilindro*, pois discos maiores têm mais de um prato, todos empilhados e girando em torno do mesmo eixo. Cada prato pode ter um ou dois cabeçotes, para a parte superior e/ou inferior do prato, e todos os cabeçotes estão conectados ao mesmo braço e se movem em conjunto. Pelo fato de o braço mover, há vários cilindros no disco, desde os pequenos em torno do centro até os maiores em torno da periferia do disco. Por fim, um cilindro pode ser dividido em fatias chamadas *setores*. Essa maneira de pensar na geometria do disco chama-se *CHS* (Cylinder-Head-Sector, ou cilindro-cabeçote-setor).

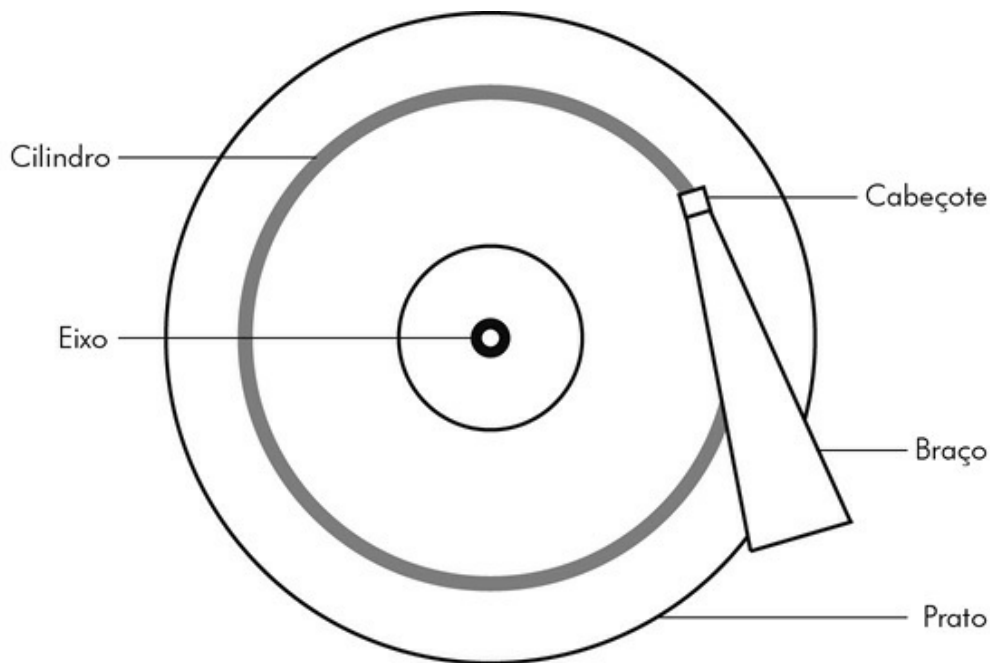


Figura 4.3 – Visão de cima para baixo de um disco rígido.

👍 **Observação:** uma trilha é uma parte de um cilindro acessada por um único cabeçote, portanto, na figura 4.3, um cilindro também é uma trilha. Provavelmente, você não precisará se preocupar com trilhas.

O kernel e os vários programas de particionamento podem dizer o que um disco informa como sendo a sua quantidade de cilindros (e de *setores*, que são as fatias dos cilindros). Entretanto, em um disco rígido moderno, *os valores informados são fictícios!* O esquema tradicional de endereçamento que usa CHS não se estende a hardwares modernos de disco, nem leva em conta o fato de que você pode colocar mais dados em cilindros mais externos em relação aos cilindros mais internos. O hardware dos discos suporta *LBA* (Logical Block Addressing, ou Endereçamento de bloco lógico) para simplesmente endereçar um local do disco por meio de um número de bloco, porém remanescentes do CHS continuam existindo. Por exemplo, a tabela de partição MBR contém informações de CHS bem como equivalentes LBA, e alguns boot loaders ainda são tolos o suficiente para acreditar nos valores CHS (não se preocupe – a maioria dos boot loaders Linux utiliza os valores LBA).

Apesar disso, a ideia de cilindros tem sido importante no particionamento porque os cilindros são fronteiras ideais para partições. Ler um stream de dados de um cilindro é muito rápido porque o cabeçote pode obter dados continuamente enquanto o disco gira. Uma partição organizada como um conjunto de cilindros adjacentes também permite um acesso contínuo e rápido de dados, pois o cabeçote não precisa se deslocar por uma longa distância entre os cilindros.

Alguns programas de particionamento reclamam se você não posicionar suas partições precisamente nas fronteiras entre cilindros. Ignore isso; não há muito que você possa

fazer, pois os valores CHS informados nos discos modernos simplesmente não são verdadeiros. O esquema de LBA do disco garante que suas partições estarão nos locais em que deveriam estar.

4.1.4 Discos de estado sólido (SSDs)

Os dispositivos de armazenamento sem partes móveis, como os *SSDs* (Solid-State Disks, ou Discos de estado sólido), são radicalmente diferentes dos discos giratórios no que diz respeito às características de acesso. Para eles, o acesso aleatório não é um problema, pois não há nenhum cabeçote para varrer um prato, porém determinados fatores afetam o desempenho.

Um dos fatores mais significativos a afetar o desempenho dos SSDs é o *alinhamento de partição*. Ao ler dados de um SSD, a leitura é feita em porções – normalmente, são 4096 bytes de cada vez –, e essa operação deve começar em um múltiplo desse mesmo tamanho. Portanto, se sua partição e seus dados não estiverem em uma fronteira de 4096 bytes, poderão ser necessárias duas leituras em vez de uma para operações pequenas e comuns, como a leitura do conteúdo de um diretório.

Muitos utilitários de particionamento (*parted* e *gparted*, por exemplo) incluem funcionalidades para colocar partições recém-criadas nos offsets adequados a partir do início dos discos, portanto pode ser que você jamais precise se preocupar com alinhamentos inadequados de partição. No entanto, se você estiver curioso para saber em que local suas partições começam e simplesmente quiser garantir que elas iniciem em uma fronteira, essa informação poderá ser facilmente encontrada em */sys/block*. Aqui está um exemplo para uma partição */dev/sdf2*:

```
$ cat /sys/block/sdf/sdf2/start  
1953126
```

Essa partição começa no byte de número 1.953.126 a partir do início do disco. Como esse número não é divisível por 4.096, a partição não atingirá um desempenho ótimo se estiver em um SSD.

4.2 Sistemas de arquivos

A última ligação entre o espaço de usuário e o espaço de kernel para discos normalmente é o *sistema de arquivos*; é o local em que você está acostumado a interagir ao executar comandos como *ls* e *cd*. Conforme mencionado anteriormente, o sistema de arquivos é uma espécie de banco de dados; ele fornece a estrutura para transformar um simples dispositivo de bloco na hierarquia sofisticada de arquivos e de

subdiretórios que os usuários possam entender.

Houve uma época em que os sistemas de arquivos permaneciam em discos e em outras mídias físicas usadas exclusivamente para armazenamento de dados. Entretanto a estrutura de diretórios semelhante a uma árvore e a interface de I/O dos sistemas de arquivos são bem versáteis, de modo que, atualmente, os sistemas de arquivos realizam uma diversidade de tarefas, por exemplo, prover as interfaces de sistema que vemos em `/sys` e `/proc`. Os sistemas de arquivos, em geral, também são implementados no kernel, porém a inovação do 9P do Plan 9 (<http://plan9.bell-labs.com/sys/doc/9.html>) inspirou o desenvolvimento de sistemas de arquivos no espaço de usuário. O recurso *FUSE* (File System in User Space, ou Sistema de arquivos no espaço de usuário) permite ter sistemas de arquivos no espaço de usuário no Linux.

A camada de abstração *VFS* (Virtual File System, ou Sistema de arquivos virtual) completa a implementação do sistema de arquivos. Assim como o subsistema SCSI padroniza a comunicação entre diferentes tipos de dispositivo e os comandos de controle do kernel, o VFS garante que todas as implementações de sistemas de arquivos suportem uma interface-padrão para que as aplicações do espaço de usuário acessem arquivos e diretórios da mesma maneira. A presença do VFS permite que o Linux suporte uma quantidade incrivelmente grande de sistemas de arquivos.

4.2.1 Tipos de sistemas de arquivos

O suporte do Linux aos sistemas de arquivos inclui designs nativos otimizados para o Linux, tipos estrangeiros como a família FAT do Windows, sistemas de arquivos universais como o ISO 9660 e vários outros. A lista a seguir inclui os tipos mais comuns de sistemas de arquivos para armazenamento de dados. Os nomes dos tipos, conforme reconhecidos pelo Linux, estão entre parênteses, ao lado dos nomes dos sistemas de arquivos.

- O *sistema de arquivos Fourth Extended* (ext4) é a iteração atual de uma linha de sistemas de arquivos nativos do Linux. O *sistema de arquivos Second Extended* (ext2), durante muito tempo, foi padrão em sistemas Linux, e foi inspirado nos sistemas de arquivos Unix tradicionais como o UFS (Unix File System) e o FFS (Fast File System). O *sistema de arquivos Third Extended* (ext3) adicionou um recurso de *journal* (uma pequena cache fora da estrutura normal de dados do sistema de arquivos) para melhorar a integridade dos dados e agilizar a inicialização. O sistema de arquivos ext4 representa uma melhoria incremental, com suporte a arquivos maiores em relação àquele oferecido pelo ext2 ou pelo ext3 e a uma quantidade maior de subdiretórios.

Até certo ponto, há compatibilidade com versões anteriores na série de sistemas de arquivos estendidos. Por exemplo, você pode montar sistemas de arquivos ext2 como ext3 e vice-versa, e pode montar sistemas de arquivos ext2 e ext3 como ext4, porém *não é possível* montar ext4 como ext2 ou ext3.

- O *ISO 9660* (iso9660) é um padrão para CD-ROM. A maioria dos CD-ROMs utiliza alguma variedade do padrão ISO 9660.
- Os *sistemas de arquivos FAT* (msdos, vfat, umsdos) dizem respeito aos sistemas Microsoft. O tipo msdos simples suporta a variedade bem primitiva, sem distinção de letras minúsculas e maiúsculas, nos sistemas MS-DOS. Na maioria dos sistemas de arquivos Windows modernos, você deve usar o sistema de arquivos vfat para ter acesso completo a partir do Linux. O sistema de arquivos umsdos, raramente usado, é próprio para o Linux. Ele suporta recursos do Linux, por exemplo, links simbólicos, sobre um sistema de arquivos MS-DOS.
- *HFS+* (hfsplus) é um padrão da Apple usado na maioria dos sistemas Macintosh.

Embora as séries de sistemas de arquivos Extended tenha sido perfeitamente aceitável para a maioria dos usuários casuais, muitos avanços foram feitos na tecnologia de sistema de arquivos que até mesmo o ext4 não pode utilizar por causa do requisito de compatibilidade com versões anteriores. Os avanços referem-se principalmente a melhorias quanto à escalabilidade, relacionadas a quantidades bem grandes de arquivos, arquivos grandes e cenários semelhantes. Novos sistemas de arquivos Linux, como o Btrfs, estão em desenvolvimento e poderão substituir a série Extended.

4.2.2 Criando um sistema de arquivos

Depois de ter visto o processo de particionamento descrito na seção 4.1, você estará pronto para criar sistemas de arquivos. Assim como no particionamento, faremos isso no espaço de usuário, pois um processo nesse espaço pode acessar e manipular diretamente um dispositivo de bloco. O utilitário `mkfs` pode criar vários tipos de sistemas de arquivos. Por exemplo, você pode criar uma partição ext4 em `/dev/sdf2` usando este comando:

```
# mkfs -t ext4 /dev/sdf2
```

O programa `mkfs` determina automaticamente a quantidade de blocos em um dispositivo e define alguns defaults razoáveis. A menos que você realmente saiba o que está fazendo e esteja disposto a ler a documentação em detalhes, não mude isso.

Ao criar um sistema de arquivos, o `mkfs` exibe uma saída contendo dados de diagnóstico à medida que executa, incluindo resultados que dizem respeito ao *superbloco*. O

superbloco é um componente fundamental do nível mais alto do banco de dados do sistema de arquivos e é tão importante que o `mkfs` cria vários backups dele caso o original seja destruído. Considere gravar alguns dos backups do superbloco quando o `mkfs` executar, se for necessário recuperar o superbloco em caso de haver uma falha em disco (veja a seção 4.2.11).

👍 **Aviso:** a criação de um sistema de arquivos é uma tarefa que você deverá executar somente após adicionar um disco novo ou reparticionar um disco antigo. Você deve criar um sistema de arquivos somente uma vez para cada nova partição que não tenha dados preexistentes (ou que tenha dados que você queira apagar). Criar um novo sistema de arquivos sobre um sistema existente irá efetivamente destruir os dados antigos.

O fato é que `mkfs` é apenas um frontend para uma série de programas de criação de sistema de arquivos `mkfs.fs`, em que `fs` é o tipo de sistema de arquivos. Portanto, ao executar `mkfs -t ext4`, o `mkfs`, por sua vez, executará `mkfs.ext4`.

E há ainda outros níveis de acesso indireto. Inspecione os arquivos `mkfs.*` por trás dos comandos e você verá o seguinte:

```
$ ls -l /sbin/mkfs.*
```

```
-rwxr-xr-x 1 root root 17896 Mar 29 21:49 /sbin/mkfs.bfs
-rwxr-xr-x 1 root root 30280 Mar 29 21:49 /sbin/mkfs.cramfs
lrwxrwxrwx 1 root root   6 Mar 30 13:25 /sbin/mkfs.ext2 -> mke2fs
lrwxrwxrwx 1 root root   6 Mar 30 13:25 /sbin/mkfs.ext3 -> mke2fs
lrwxrwxrwx 1 root root   6 Mar 30 13:25 /sbin/mkfs.ext4 -> mke2fs
lrwxrwxrwx 1 root root   6 Mar 30 13:25 /sbin/mkfs.ext4dev -> mke2fs
-rwxr-xr-x 1 root root 26200 Mar 29 21:49 /sbin/mkfs.minix
lrwxrwxrwx 1 root root   7 Dec 19 2011 /sbin/mkfs.msdos -> mkdosfs
lrwxrwxrwx 1 root root   6 Mar  5 2012 /sbin/mkfs.ntfs -> mkntfs
lrwxrwxrwx 1 root root   7 Dec 19 2011 /sbin/mkfs.vfat -> mkdosfs
```

Como você pode ver, `mkfs.ext4` é apenas um link simbólico para `mke2fs`. É importante lembrar-se disso se você se deparar com um sistema que não tenha um comando `mkfs` específico ou quando estiver consultando a documentação de um sistema de arquivos em particular. Cada utilitário para criação de sistema de arquivos tem sua própria página de manual, por exemplo, `mke2fs(8)`. Isso não deverá ser um problema na maioria dos sistemas, pois acessar a página de manual `mkfs.ext4(8)` deverá redirecionar você para a página de manual `mke2fs(8)`, mas tenha essa informação em mente.

4.2.3 Montando um sistema de arquivos

No Unix, o processo de fazer a associação com um sistema de arquivos chama-se *montagem*. Quando o sistema é inicializado, o kernel lê alguns dados de configuração e monta a raiz (`/`) de acordo com esses dados.

Para montar um sistema de arquivos, você deve saber o seguinte:

- O dispositivo relacionado ao sistema de arquivos (por ex., uma partição de disco; é o local em que estão os dados propriamente ditos do sistema de arquivos).
- O tipo do sistema de arquivos.
- O *ponto de montagem* (mount point), ou seja, o local na hierarquia de diretórios do sistema atual ao qual o sistema de arquivos será associado. O ponto de montagem é sempre um diretório normal. Por exemplo, você pode usar */cdrom* como um ponto de montagem para dispositivos de CD-ROM. O ponto de montagem não precisa estar diretamente abaixo de */*; ele poderá estar em qualquer local no sistema.

Ao montar um sistema de arquivos, a terminologia comum consiste em dizer “montar um dispositivo *em* um ponto de montagem”. Para saber o status atual dos sistemas de arquivos em seu sistema, execute `mount`. A saída deverá ser semelhante a:

```
$ mount
/dev/sda1 on / type ext4 (rw,errors=remount-ro)
proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
none on /sys/fs/fuse/connections type fusectl (rw)
none on /sys/kernel/debug type debugfs (rw)
none on /sys/kernel/security type securityfs (rw)
udev on /dev type devtmpfs (rw,mode=0755)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
--trecho omitido--
```

Cada linha corresponde a um sistema de arquivos montado no momento, com itens nesta ordem:

- O dispositivo, por exemplo */dev/sda3*. Observe que alguns deles não são dispositivos de verdade (*proc*, por exemplo), mas substitutos para nomes reais de dispositivos, pois esses sistemas de arquivos de propósito especial não precisam de dispositivos.
- A palavra *on*.
- O ponto de montagem.
- A palavra *type*.
- O tipo de sistema de arquivos, normalmente na forma de um identificador compacto.
- As opções de montagem (entre parênteses). (Veja a seção 4.2.6 para obter mais detalhes.)

Para montar um sistema de arquivos, utilize o comando `mount` como se segue, com o tipo

do sistema de arquivos, o dispositivo e o ponto de montagem desejado:

```
# mount -t tipo dispositivo pontodemontagem
```

Por exemplo, para montar o sistema de arquivos Fourth Extended *dev/sdf2* em */home/extra*, utilize o comando a seguir:

```
# mount -t ext4 /dev/sdf2 /home/extra
```

Normalmente, não será necessário fornecer a opção *-t tipo* porque *mount*, em geral, poderá descobrir isso para você. Porém, às vezes, é necessário fazer a distinção entre dois tipos semelhantes, por exemplo, entre os vários sistemas de arquivos de estilo FAT.

Consulte a seção 4.2.6 para conhecer mais algumas opções longas de montagem. Para desmontar (desassociar) um sistema de arquivos, utilize o comando *umount*:

```
# umount pontodemontagem
```

Você também pode desmontar um sistema de arquivos usando o seu dispositivo em vez de utilizar o ponto de montagem.

4.2.4 UUID de sistemas de arquivos

O método de montagem de sistemas de arquivos discutido na seção anterior depende dos nomes dos dispositivos. Entretanto os nomes dos dispositivos podem mudar, pois dependem da ordem em que o kernel os encontra. Para solucionar esse problema, você pode identificar e montar sistemas de arquivos de acordo com seus UUID (Universally Unique Identifier, ou Identificador universal único) – um padrão de software. O UUID é um tipo de número de série, e cada número deve ser diferente. Os programas de criação de sistemas de arquivos, como o *mke2fs*, geram um identificador UUID ao inicializarem a estrutura de dados do sistema de arquivos.

Para ver uma lista dos dispositivos, os sistemas de arquivos correspondentes e os UUIDs em seu sistema, utilize o programa *blkid* (block ID):

```
# blkid
/dev/sdf2: UUID="a9011c2b-1c03-4288-b3fe-8ba961ab0898" TYPE="ext4"
/dev/sda1: UUID="70ccd6e7-6ae6-44f6-812c-51aab8036d29" TYPE="ext4"
/dev/sda5: UUID="592dcfd1-58da-4769-9ea8-5f412a896980" TYPE="swap"
/dev/sde1: SEC_TYPE="msdos" UUID="3762-6138" TYPE="vfat"
```

Nesse exemplo, *blkid* encontrou quatro partições com dados: duas com sistemas de arquivos *ext4*, uma com uma assinatura de espaço de swap (veja a seção 4.3) e uma com um sistema de arquivos baseado em FAT. Todas as partições nativas do Linux têm UUIDs-padrão, porém a partição FAT não tem. Você pode se referir à partição FAT por

meio de seu número de série de volume FAT (nesse caso, é 3762-6138).

Para montar um sistema de arquivos de acordo com seu UUID, utilize a sintaxe `UUID=`. Por exemplo, para montar o primeiro sistema de arquivos da lista anterior em `/home/extra`, digite:

```
# mount UUID=a9011c2b-1c03-4288-b3fe-8ba961ab0898 /home/extra
```

Normalmente, você não irá montar sistemas de arquivos manualmente pelo UUID, como foi feito anteriormente, pois é provável que você conhecerá o dispositivo, e é muito mais fácil montar um dispositivo pelo seu nome do que pelo número de UUID maluco. Entretanto é importante entender os UUIDs. Um dos motivos é porque eles são a maneira preferida de montar sistemas de arquivos automaticamente em `/etc/fstab` no momento do boot (veja a seção 4.2.8). Além do mais, muitas distribuições usam o UUID como um ponto de montagem quando uma mídia removível é inserida. No exemplo anterior, o sistema de arquivos FAT está em um cartão de mídia flash. Um sistema Ubuntu que tiver um usuário logado irá montar essa partição em `/media/3762-6138` quando o dispositivo for inserido. O daemon `udev` descrito no capítulo 3 cuida do evento inicial na inserção do dispositivo.

O UUID de um sistema de arquivos pode ser alterado se for necessário (por exemplo, se você copiou todo o sistema de arquivos de outro local e agora precisa distingui-lo em relação ao original). Consulte a página de manual `tune2fs(8)` para saber como fazer isso em um sistema de arquivos `ext2/ext3/ext4`.

4.2.5 Bufferização em disco, caching e sistemas de arquivos

O Linux, assim como outras versões de Unix, bufferiza as escritas em disco. Isso significa que o kernel normalmente não escreve de imediato as alterações nos sistemas de arquivos quando os processos solicitam alterações. Em vez disso, ele armazena as alterações em RAM até o kernel poder fazer realmente as alterações em disco de forma conveniente. Esse sistema de bufferização é transparente para o usuário e melhora o desempenho.

Ao desmontar um sistema de arquivos com `umount`, o kernel efetua uma sincronização automática com o disco. A qualquer momento, você pode forçar o kernel a gravar as alterações de seu buffer para o disco ao executar o comando `sync`. Se, por algum motivo, você não puder desmontar um sistema de arquivos antes de desligar o sistema, não se esqueça de executar `sync` antes.

Além do mais, o kernel tem uma série de mecanismos que usam a RAM para fazer cache automaticamente de blocos lidos de um disco. Sendo assim, se um ou mais

processos acessarem repetidamente um arquivo, o kernel não precisará acessar o disco repetidamente – ele poderá simplesmente ler da cache e economizar tempo e recursos.

4.2.6 Opções de montagem de sistemas de arquivos

Há várias maneiras de mudar o comportamento do comando `mount`, como frequentemente é necessário com mídias removíveis ou quando você estiver realizando manutenção no sistema. Com efeito, a quantidade total de opções de `mount` é impressionante. A extensa página de manual `mount(8)` é uma boa referência, porém é difícil saber em que ponto começar e o que pode ser ignorado de forma segura. Você verá as opções mais úteis nesta seção.

As opções se enquadram em duas categorias genéricas: as opções gerais e as opções específicas de sistemas de arquivos. As opções gerais incluem `-t` para especificar o tipo do sistema de arquivos (conforme mencionado anteriormente). Em contrapartida, uma opção específica a um sistema de arquivos diz respeito somente a determinados tipos de sistema de arquivos.

Para ativar uma opção de sistema de arquivos, utilize a opção `-o`, seguida da opção. Por exemplo, `-o norock` desabilita as extensões Rock Ridge em um sistema de arquivos ISO 9660, porém não faz sentido para qualquer outro tipo de sistema de arquivos.

Opções curtas

As opções gerais mais importantes são estas:

- `-r` – a opção `-r` monta o sistema de arquivos em modo somente de leitura. Essa opção tem vários propósitos, desde a proteção contra escrita até o bootstrapping (inicialização do sistema). Não é preciso especificar essa opção ao acessar um dispositivo somente de leitura como um CD-ROM; o sistema fará isso por você (e também irá informá-lo sobre o status somente de leitura).
- `-n` – a opção `-n` garante que `mount` não tentará atualizar o banco de dados de montagem do sistema criado em tempo de execução, o `/etc/mtab`. A operação `mount` falha quando não consegue escrever nesse arquivo, o que é importante no momento do boot, pois a partição raiz (e, desse modo, o banco de dados de montagem do sistema), inicialmente, é somente para leitura. Você também achará essa opção prática quando estiver tentando corrigir um problema de sistema em modo monousuário (single-user mode), pois o banco de dados de montagem do sistema poderá não estar disponível nesse momento.
- `-t` – a opção `-t tipo` especifica o tipo do sistema de arquivos.

Opções longas

As opções curtas, como `-r`, são limitadas demais para a quantidade cada vez maior de opções de `mount`; há poucas letras no alfabeto para acomodar todas as opções possíveis. As opções curtas também são problemáticas porque é difícil determinar o significado de uma opção com base em apenas uma letra. Muitas opções gerais e todas as opções específicas de sistemas de arquivos usam um formato de opção mais longo e mais flexível.

Para usar opções longas com `mount` na linha de comando, comece com `-o` e forneça algumas palavras-chave. Aqui está um exemplo completo, com opções longas após `-o`:

```
# mount -t vfat /dev/hda1 /dos -o ro,conv=auto
```

As duas opções longas, nesse caso, são `ro` e `conv=auto`. A opção `ro` especifica o modo somente de leitura e é igual à opção curta `-r`. A opção `conv=auto` diz ao kernel para converter automaticamente determinados arquivos texto do formato de quebra de linha do DOS para o estilo Unix (você verá mais sobre esse assunto em breve).

As opções longas mais úteis são:

- `exec`, `noexec` — habilita ou desabilita a execução de programas no sistema de arquivos.
- `suid`, `nosuid` — habilita ou desabilita programas setuid.
- `ro` — monta o sistema de arquivos em modo somente de leitura (como faz a opção curta `-r`).
- `rw` — monta o sistema de arquivos em modo de leitura e escrita.
- `conv=regra` — (sistemas de arquivos baseados em FAT) converte caracteres de quebra de linha em arquivos de acordo com *regra*, que pode ser `binary`, `text` ou `auto`. O default é `binary`, que desabilita qualquer tradução de caracteres. Para tratar todos os arquivos como texto, use `text`. A configuração `auto` converte os arquivos de acordo com suas extensões. Por exemplo, um arquivo `.jpg` não recebe nenhum tratamento especial, porém um arquivo `.txt` recebe. Tome cuidado com essa opção, pois ela pode danificar os arquivos. Considere usá-la em modo somente de leitura.

4.2.7 Remontando um sistema de arquivos

Haverá ocasiões em que poderá ser necessário reconectar um sistema de arquivos já montado no mesmo ponto de montagem, quando for preciso alterar opções de montagem. A situação mais comum é aquela em que é preciso permitir a escrita em um sistema de arquivos somente de leitura durante uma recuperação de falha.

O comando a seguir remonta a raiz em modo somente de leitura (a opção `-n` é

necessária, pois o comando `mount` não pode escrever no banco de dados de montagem do sistema quando a raiz for somente para leitura):

```
# mount -n -o remount /
```

Esse comando supõe que a listagem correta de dispositivos para `/` está em `/etc/fstab` (conforme discutido na próxima seção). Se não estiver, você deverá especificar o dispositivo.

4.2.8 A tabela de sistemas de arquivos `/etc/fstab`

Para montar os sistemas de arquivos na inicialização do sistema e evitar o trabalho pesado para o comando `mount`, os sistemas Linux mantêm uma lista permanente de sistemas de arquivos e de opções em `/etc/fstab`. É um arquivo texto em um formato bem simples, como mostrado na listagem 4.1.

Listagem 4.1 – Lista de sistemas de arquivos e as opções em `/etc/fstab`

```
proc /proc proc nodev,noexec,nosuid 0 0
UUID=70ccd6e7-6ae6-44f6-812c-51aab8036d29 / ext4 errors=remount-ro 0 1
UUID=592dcfd1-58da-4769-9ea8-5f412a896980 none swap sw 0 0
/dev/sr0 /cdrom iso9660 ro,user,nosuid,noauto 0 0
```

Cada linha corresponde a um sistema de arquivos, cada qual dividido em seis campos. Esses campos estão descritos a seguir, na sequência, da esquerda para a direita:

- dispositivo ou o UUID – a maioria dos sistemas Linux atuais não usa mais o dispositivo em `/etc/fstab`, preferindo utilizar o UUID. (Observe que a entrada `/proc` tem um dispositivo substituto chamado `proc`.)
- ponto de montagem – indica em que local o sistema de arquivos deve ser associado.
- tipo do sistema de arquivos – você poderá não reconhecer `swap` nessa lista; essa é uma partição de swap (veja a seção 4.3).
- opções – use opções longas, separadas por vírgulas.
- informações de backup usadas pelo comando `dump` – sempre use 0 nesse campo.
- ordem de teste de integridade do sistema de arquivos – para garantir que `fsck` sempre executará na raiz primeiro, sempre configure esse campo com 1 para o sistema de arquivos raiz e 2 para qualquer outro sistema de arquivos em um disco rígido. Use 0 para desabilitar a verificação na inicialização do sistema para tudo o

mais, incluindo drives de CD-ROM, swap e o sistema de arquivos */proc* (veja o comando `fsck` na seção 4.2.11).

Ao usar `mount`, você poderá usar alguns atalhos se o sistema de arquivos com o qual você quiser trabalhar estiver em */etc/fstab*. Por exemplo, se estivesse usando a listagem 4.1 e montando um CD-ROM, você poderia simplesmente executar `mount /cdrom`.

Você também poderá tentar montar todas as entradas de uma só vez em */etc/fstab* que não contenham a opção `noauto` usando este comando:

```
# mount -a
```

A listagem 4.1 contém algumas opções novas, que são `errors`, `noauto` e `user`, que não se aplicam fora do arquivo */etc/fstab*. Além do mais, com frequência, você verá a opção `defaults` aqui. Os significados dessas opções estão sendo apresentados a seguir:

- `defaults` — utiliza os defaults de `mount`, ou seja, usa o modo de leitura e escrita, habilita arquivos de dispositivo, os executáveis, o bit `setuid` e assim por diante. Use isso quando não quiser fornecer nenhuma opção especial ao sistema de arquivos, porém quiser preencher todos os campos em */etc/fstab*.
- `errors` — esse parâmetro específico de `ext2` define o comportamento do kernel quando o sistema tiver problemas para montar um sistema de arquivos. O default normalmente é `errors=continue`, o que significa que o kernel deverá retornar um código de erro e continuar executando. Para fazer o kernel tentar a montagem novamente em modo somente de leitura, use `errors=remount-ro`. A configuração `errors=panic` diz ao kernel (e ao seu sistema) para parar quando houver um problema com a montagem.
- `noauto` — essa opção diz a um comando `mount -a` para ignorar a entrada. Uso-a para evitar a montagem de um dispositivo de mídia removível no boot, por exemplo, um drive de CD-ROM ou de disquete.
- `user` — essa opção permite que usuários que não sejam privilegiados executem `mount` em uma entrada em particular, o que pode ser prático para permitir acesso a drives de CD-ROM. Como os usuários podem colocar um arquivo `setuid-root` em uma mídia removível com outro sistema, essa opção também define `nosuid`, `noexec` e `nODEV` (para barrar arquivos especiais de dispositivo).

4.2.9 Alternativas para */etc/fstab*

Embora o arquivo */etc/fstab* tenha sido a maneira tradicional de representar sistemas de arquivos e seus pontos de montagem, duas novas alternativas surgiram. A primeira é um diretório */etc/fstab.d* que contém arquivos individuais de configuração de sistemas de arquivos (um arquivo para cada sistema de arquivos). A ideia é bem semelhante àquela

usada em vários outros diretórios de configuração que você verá ao longo deste livro.

Uma segunda alternativa consiste em configurar unidades `systemd` para os sistemas de arquivos. Você aprenderá mais sobre `systemd` e suas unidades no capítulo 6. No entanto a configuração de unidades do `systemd` geralmente é gerada a partir do (ou de acordo com o) arquivo `/etc/fstab`, portanto você poderá encontrar um pouco de sobreposição em seu sistema.

4.2.10 Capacidade dos sistemas de arquivos

Para ver o tamanho e a utilização dos seus sistemas de arquivos montados no momento, utilize o comando `df`. A saída deverá ser semelhante a:

```
$ df
Filesystem 1024-blocks  Used Available Capacity Mounted on
/dev/sda1   1011928  71400   889124    7%  /
/dev/sda3   17710044 9485296  7325108   56%  /usr
```

A seguir, apresentamos uma breve descrição dos campos da saída de `df`:

- **Filesystem** (sistema de arquivos) – o dispositivo do sistema de arquivos;
- **1024-blocks** (blocos de 1024) – a capacidade total do sistema de arquivos em blocos de 1024 bytes;
- **Used** (usados) – a quantidade de blocos ocupados;
- **Available** (disponível) – a quantidade de blocos livres;
- **Capacity** (capacidade) – o percentual de blocos em uso;
- **Mounted on** (montado em) – o ponto de montagem.

Deve ser fácil perceber que os dois sistemas de arquivos nesse caso têm aproximadamente 1 GB e 17,5 GB de tamanho. Entretanto os números referentes à capacidade podem parecer um pouco estranhos, pois 71.400 mais 889.124 não é igual a 1.011.928, e 9.485.296 não constitui 56% de 17.710.044. Em ambos os casos, 5% da capacidade total não foi levada em consideração. Com efeito, o espaço está presente, porém está oculto em blocos *reservados*. Desse modo, somente o superusuário pode usar o espaço total do sistema de arquivos se o restante da partição for ocupado. Esse recurso evita que os servidores do sistema falhem imediatamente quando ficarem sem espaço em disco.

Se o seu disco encher e for necessário saber o local em que estão todos aqueles arquivos de mídia consumidores de espaço, utilize o comando `du`. Sem argumentos, `du` exibe o uso de disco para todos os diretórios na hierarquia de diretórios, começando

pelo diretório de trabalho corrente. (É uma quantidade exagerada de informações, portanto execute somente `cd /; du` para ter uma ideia. Tecle Ctrl-C quando se sentir entediado.) O comando `du -s` ativa o modo resumido para exibir somente o total geral. Para avaliar um diretório em particular, vá para esse diretório e execute `du -s *`.

👉 **Observação:** o padrão POSIX define um tamanho de bloco igual a 512 bytes. Entretanto esse tamanho é mais difícil de ler, portanto, por padrão, a saída de `df` e de `du` na maioria das distribuições Linux é apresentada em blocos de 1024 bytes. Se você insistir em exibir os números em blocos de 512 bytes, defina a variável de ambiente `POSIXLY_CORRECT`. Para especificar explicitamente blocos de 1024 bytes, utilize a opção `-k` (ambos os utilitários a suportam). O programa `df` também tem uma opção `-m` para listar as capacidades em blocos de 1 MB, e uma opção `-h` para dar o melhor palpite possível sobre o que uma pessoa pode ler.

4.2.11 Verificando e fazendo reparos em sistemas de arquivos

As otimizações que os sistemas de arquivos Unix oferecem são possíveis devido a um sistema sofisticado de banco de dados. Para que os sistemas de arquivos funcionem naturalmente, o kernel deve acreditar que não há erros em um sistema de arquivos montado. Se houver erros, perda de dados e falhas no sistema poderão ocorrer.

Os erros em sistemas de arquivos normalmente ocorrem como resultado de um usuário desligar o sistema de modo grosseiro (por exemplo, puxando o fio de alimentação da tomada). Em casos como esse, a cache do sistema de arquivos em memória poderá não corresponder aos dados em disco, e o sistema também poderia estar no processo de alterar o sistema de arquivos quando você resolveu desligar o computador indevidamente. Embora uma nova geração de sistemas de arquivos suporte *journals* para que seja bem menos comum que o sistema de arquivos seja corrompido, sempre desligue o sistema de maneira apropriada. E, independentemente do sistema de arquivos em uso, verificações do sistema de arquivos continuam sendo necessárias ocasionalmente para mantê-lo saudável.

A ferramenta para verificar um sistema de arquivos é o `fsck`. Assim como para o programa `mkfs`, há uma versão diferente de `fsck` para cada tipo de sistema de arquivos suportado pelo Linux. Por exemplo, ao executar `fsck` em um sistema de arquivos da série Extended (`ext2/ext3/ext4`), esse reconhece o tipo de sistema de arquivos e inicia o utilitário `e2fsck`. Desse modo, em geral, não é preciso digitar `e2fsck`, a menos que `fsck` não consiga descobrir o tipo do sistema de arquivos ou que você esteja procurando a página de manual do `e2fsck`.

As informações apresentadas nesta seção são específicas da série de sistema de arquivos Extended e de `e2fsck`.

Para executar `fsck` em modo interativo e manual, forneça o dispositivo ou o ponto de montagem (conforme listado em `/etc/fstab`) como argumento. Por exemplo:

`fsck /dev/sdb1`

👍 **AVISO:** jamais use `fsck` em um sistema de arquivos montado porque o kernel poderá alterar os dados do disco à medida que a verificação for executada, causando erros em tempo de execução que poderão provocar falhas em seu sistema e corromper os arquivos. Há somente uma exceção: se você montar a partição raiz somente para leitura em modo monousuário, `fsck` poderá ser executado nela.

Em modo manual, `fsck` exibe informações extensas de status em suas passagens, que deverão ter uma aparência como a que se segue quando não houver problemas:

Pass 1: Checking inodes, blocks, and sizes

Pass 2: Checking directory structure

Pass 3: Checking directory connectivity

Pass 4: Checking reference counts

Pass 5: Checking group summary information

/dev/sdb1: 11/1976 files (0.0% non-contiguous), 265/7891 blocks

Se `fsck` encontrar um problema em modo manual, ele irá parar e fará uma pergunta relevante à correção do problema. Essas perguntas estão relacionadas com a estrutura interna do sistema de arquivos, como a reconexão de inodes perdidos e a limpeza de blocos (um inode é um bloco de construção do sistema de arquivos; você conhecerá o funcionamento dos inodes na seção 4.5). Quando o `fsck` perguntar sobre a reconexão de um inode, é porque ele encontrou um arquivo que parece não ter um nome. Ao reconectar um arquivo como esse, `fsck` colocará o arquivo no diretório *lost+found* do sistema de arquivos, com um número como o nome do arquivo. Se isso ocorrer, será preciso adivinhar o nome de acordo com o conteúdo do arquivo; o nome original provavelmente terá sido perdido.

Em geral, não fará sentido esperar todo o processo de reparação do `fsck`, se você acabou de desligar o sistema de forma inadequada, pois o `fsck` poderá ter muitos erros menores para corrigir. Felizmente, o `e2fsck` tem uma opção `-p` que corrige automaticamente problemas comuns sem perguntar e aborta quando houver um erro sério. Com efeito, as distribuições Linux executam alguma variante de `fsck -p` durante o boot. (Você também poderá ver `fsck -a`, que faz exatamente o mesmo.)

Se você suspeita que houve um grande desastre em seu sistema, por exemplo, uma falha de hardware ou um erro de configuração de dispositivo, será preciso decidir o curso de ação a ser tomado, pois o `fsck` pode realmente bagunçar um sistema de arquivos que tenha problemas mais sérios. (Um sinal evidente de que o seu sistema tem um problema sério é o fato de o `fsck` fazer *muitas* perguntas em modo manual.)

Se você crê que algo realmente ruim aconteceu, procure executar `fsck -n` para verificar o sistema de arquivos sem modificar nada. Se houver um problema com a configuração do dispositivo que você ache que possa ser corrigido (por exemplo, uma quantidade

incorreta de blocos na tabela de partição ou cabos soltos), corrija isso antes de realmente executar `fsck`, ou é provável que você vá perder muitos dados.

Se você suspeita que somente o superbloco está corrompido (por exemplo, porque alguém escreveu no início da partição de disco), será possível recuperar o sistema de arquivos com um dos backups do superbloco criados por `mkfs`. Use `fsck -b num` para substituir o superbloco corrompido por uma alternativa no bloco `num` e espere pelo melhor.

Se não souber em que local um backup do superbloco pode ser encontrado, você poderá executar `mkfs -n` no dispositivo para ver uma lista dos números de backups do superbloco sem destruir seus dados. (Novamente, *não se esqueça* de usar `-n`; do contrário, você *realmente* irá destruir o sistema de arquivos.)

Verificando sistemas de arquivos ext3 e ext4

Normalmente, não será necessário verificar sistemas de arquivos ext3 e ext4 manualmente porque o journal garante a integridade dos dados. Entretanto pode ser que você queira montar um sistema de arquivos ext3 ou ext4 com problemas em modo ext2, pois o kernel não montará um sistema de arquivos ext3 ou ext4 com um *journal* que não esteja vazio. (Se o seu sistema não for desligado adequadamente, você poderá esperar que o *journal* contenha alguns dados.) Para limpar o *journal* em um sistema de arquivos ext3 ou ext4 para o banco de dados normal do sistema de arquivos, execute `e2fsck` da seguinte maneira:

```
# e2fsck -fy /dev/dispositivo_de_disco
```

O pior caso

Problemas em disco que sejam piores quanto à severidade deixam você com poucas opções:

- Você pode tentar extrair a imagem completa do sistema de arquivos do disco usando `dd` e transferi-la para uma partição em outro disco de mesmo tamanho.
- Você pode tentar corrigir o sistema de arquivos o máximo possível, montá-lo em modo somente de leitura e salvar o que puder.
- Você pode tentar o `debugfs`.

Nos dois primeiros casos, ainda será preciso corrigir o sistema de arquivos antes de montá-lo, a menos que você esteja disposto a acessar manualmente os dados brutos. Se quiser, você poderá optar por responder `y` a todas as perguntas de `fsck` digitando `fsck -y`, porém faça isso como último recurso, pois poderão ocorrer problemas durante o

processo de reparação com os quais você preferirá lidar manualmente.

A ferramenta `debugfs` permite observar os arquivos em um sistema de arquivos e copiá-los para outro lugar. Por padrão, ele abre os sistemas de arquivos em modo somente de leitura. Se você estiver recuperando dados, provavelmente será uma boa ideia manter seus arquivos intactos para evitar um estrago maior ainda.

Porém, se você estiver realmente desesperado, por exemplo, com uma falha catastrófica de disco em suas mãos, sem backup, não há muito que fazer além de esperar que um serviço profissional consiga “raspar os pratos”.

4.2.12 Sistemas de arquivos com propósitos especiais

Nem todos os sistemas de arquivos representam área de armazenamento em mídias físicas. Especificamente, a maioria das versões de Unix tem sistemas de arquivos que servem como interfaces do sistema. Isso quer dizer que, em vez de servir somente como um meio para armazenar dados em um dispositivo, um sistema de arquivos pode representar informações do sistema, como IDs de processo e diagnósticos do kernel. Essa ideia remonta ao sistema `/dev`, que é um modelo antigo de uso de arquivos para interfaces de I/O. A ideia do `/proc` surgiu da oitava edição do Unix para pesquisa, implementada por Tom J. Killian e agilizada quando o Bell Labs (incluindo muitos dos designers originais do Unix) criou o Plan 9 – um sistema operacional de pesquisa que levou a abstração do sistema de arquivos a um nível totalmente novo (<http://plan9.bell-labs.com/sys/doc/9.html>).

Os tipos especiais de sistemas de arquivos de uso comum no Linux incluem:

- `proc` – montado em `/proc`. O nome *proc* na realidade é uma abreviação de *process* (processo). Cada diretório *numerado* em `/proc` é o ID de processo de um processo corrente no sistema; os arquivos nesses diretórios representam vários aspectos dos processos. O arquivo `/proc/self` representa o processo corrente. O sistema de arquivos `proc` do Linux inclui uma grande quantidade de informações adicionais do kernel e de hardware em arquivos como `/proc/cpuinfo`. (Tem havido pressões para transferir as informações não relacionadas a processos para `/sys`, fora de `/proc`.)
- `sysfs` – montado em `/sys`. (Você o viu no capítulo 3.)
- `tmpfs` – montado em `/run` e em outros locais. Com o `tmpfs`, você pode usar sua memória física e o espaço de swap como áreas de armazenamento temporárias. Por exemplo, você pode montar `tmpfs` nos locais em que quiser usando as opções longas `size` e `nr_blocks` para controlar o tamanho máximo. No entanto tome cuidado para não colocar itens constantemente em um `tmpfs` porque o seu sistema, em algum momento,

ficará sem memória e os programas começarão a provocar falhas. (Durante anos, a Sun Microsystems usou uma versão de tmpfs para */tmp* que causava problemas em sistemas que executassem por longos períodos.)

4.3 Espaço de swap

Nem toda partição em um disco contém um sistema de arquivos. Também é possível aumentar a RAM em um computador usando espaço em disco. Se você ficar sem memória real, o sistema de memória virtual do Linux poderá transferir partes da memória automaticamente de e para uma área de armazenamento em disco. Isso é chamado de *swapping* (troca) porque partes de programas inativos são passadas para o disco em troca de partes ativas que estão no disco. A área de disco usada para armazenar páginas de memória chama-se *espaço de swap* (ou simplesmente *swap* para abreviar).

A saída do comando `free` inclui o uso corrente de swap em kilobytes como se segue:

```
$ free
      total    used    free
--trecho omitido--
Swap:   514072   189804   324268
```

4.3.1 Usando uma partição de disco como área de swap

Para usar toda uma partição de disco como swap, siga estes passos:

1. Certifique-se de que a partição esteja vazia.
2. Execute `mkswap disp`, em que *disp* é o dispositivo referente à partição. Esse comando coloca uma assinatura de swap na partição.
3. Execute `swapon disp` para registrar o espaço junto ao kernel.

Após ter criado uma partição de swap, você poderá colocar uma nova entrada de swap em seu arquivo */etc/fstab* para fazer o sistema usar o espaço de swap assim que o computador for inicializado. A seguir, apresentamos uma entrada de exemplo que usa */dev/sda5* como partição de swap:

```
/dev/sda5 none swap sw 0 0
```

Tenha em mente que muitos sistemas atualmente usam UUIDs em vez de utilizar nomes de dispositivo.

4.3.2 Usando um arquivo como área de swap

Um arquivo normal pode ser usado como área de swap se você estiver em uma situação

em que seja forçado a refazer a partição de um disco para criar uma partição de swap. Você não deverá ter nenhum problema ao fazer isso.

Utilize os comandos a seguir para criar um arquivo vazio, inicialize-o como swap e adicione-o ao pool de swap:

```
# dd if=/dev/zero of=arquivo_de_swap bs=1024k count=num_mb  
# mkswap arquivo_de_swap  
# swapon arquivo_de_swap
```

Nesse caso, *arquivo_de_swap* é o nome do novo arquivo de swap, e *num_mb* é o tamanho desejado em megabytes.

Para remover uma partição ou um arquivo de swap do pool ativo do kernel, utilize o comando `swapoff`.

4.3.3 De que quantidade de swap você precisa?

Houve época em que, de acordo com a sabedoria convencional do Unix, você deveria reservar uma área de swap correspondente a pelo menos o dobro da memória real. Atualmente, não só as enormes capacidades de disco e de memória disponíveis como também as maneiras como usamos o sistema eclipsam o problema. Por um lado, o espaço em disco é tão abundante que é tentador alocar mais que o dobro do tamanho da memória. Por outro, pode ser que você jamais entre em contato com sua área de swap porque há bastante memória real.

A regra do “dobro da memória real” data da época em que vários usuários faziam login em um único computador ao mesmo tempo. Nem todos, porém, estavam ativos, portanto era conveniente poder fazer swap da memória dos usuários inativos quando um usuário ativo precisasse de mais memória.

O mesmo pode continuar valendo para um computador monousuário. Se você estiver executando diversos processos, em geral, é bom fazer swap de partes dos processos inativos ou até mesmo de partes inativas de processos ativos. No entanto, se você estiver usando constantemente a área de swap porque muitos processos ativos querem usar a memória ao mesmo tempo, você terá sérios problemas de desempenho, pois o I/O de disco é simplesmente lento demais para acompanhar o restante do sistema. As únicas soluções são comprar mais memória, finalizar alguns processos ou reclamar.

Às vezes, o kernel do Linux pode optar por fazer swap de um processo em favor de um pouco mais de cache em disco. Para evitar esse comportamento, alguns administradores configuram determinados sistemas sem nenhuma área de swap. Por exemplo, servidores de rede de alto desempenho jamais devem usar áreas de swap e devem evitar acessos a

disco, se for possível.

 **Observação:** é perigoso fazer isso em um computador de propósito geral. Se um computador ficar totalmente sem memória real e área de swap, o kernel do Linux chamará o killer OOM (out-of-memory) para matar um processo de modo a disponibilizar um pouco de memória. Obviamente, você não vai querer que isso aconteça com suas aplicações desktop. Por outro lado, servidores de alto desempenho incluem sistemas sofisticados de monitoração e de balanceamento de carga para garantir que eles jamais atinjam a zona de perigo.

Você aprenderá mais sobre o funcionamento do sistema de memória no capítulo 8.

4.4 Próximos passos: discos e espaço de usuário

Em componentes relacionados a discos em um sistema Unix, as fronteiras entre espaço de usuário e espaço de kernel podem ser difíceis de ser caracterizadas. Como você viu, o kernel lida com I/O de blocos brutos dos dispositivos, e as ferramentas de espaço de usuário podem utilizar I/O de blocos por meio de arquivos de dispositivo. Entretanto o espaço de usuário normalmente usa I/O de blocos somente para inicializar operações como particionamento, criação de sistemas de arquivos e de espaço de swap. Em uso normal, o espaço de usuário utiliza somente o suporte a sistemas de arquivos disponibilizado pelo kernel sobre o I/O de blocos. De modo semelhante, o kernel também cuida da maioria dos detalhes tediosos ao lidar com o espaço de swap no sistema de memória virtual.

O restante deste capítulo discute brevemente os detalhes internos de um sistema de arquivos Linux. É um material mais avançado, e você, certamente, não precisará conhecê-lo para prosseguir com o livro. Se essa, porém, for a sua primeira vez no assunto, pule para o próximo capítulo e comece a aprender sobre o modo como o Linux inicializa.

4.5 Dentro de um sistema de arquivos tradicional

Um sistema de arquivos Unix tradicional tem dois componentes principais: um pool de blocos de dados em que você pode armazenar dados e um sistema de banco de dados que administra o pool de dados. O banco de dados está centrado em torno da estrutura de dados inode. Um *inode* é um conjunto de dados que descreve um arquivo em particular, incluindo o seu tipo, as permissões e – talvez o mais importante – em que local no pool de dados estão os dados do arquivo. Os inodes são identificados por números listados em uma tabela de inodes.

Nomes de arquivos e diretórios também são implementados como inodes. Um inode de diretório contém uma lista de nomes de arquivos e os links correspondentes para outros

inodes.

Para oferecer um exemplo da vida real, criei um novo sistema de arquivos, fiz a sua montagem e mudei de diretório para o ponto de montagem. Em seguida, adicionei alguns arquivos e diretórios usando os comandos a seguir (sinta-se à vontade para fazer isso por conta própria em um pen drive):

```
$ mkdir dir_1
$ mkdir dir_2
$ echo a > dir_1/file_1
$ echo b > dir_1/file_2
$ echo c > dir_1/file_3
$ echo d > dir_2/file_4
$ ln dir_1/file_3 dir_2/file_5
```

Observe que criei *dir_2/file_5* como um hard link para *dir_1/file_3*, o que significa que esses dois nomes de arquivo na realidade representam o mesmo arquivo. (Mais sobre esse assunto em breve.)

Se você explorar os diretórios nesse sistema de arquivos, seu conteúdo aparecerá ao usuário conforme mostrado na figura 4.4. O layout propriamente dito do sistema de arquivos, como mostrado na figura 4.5, não está nem perto de parecer tão claro quanto a representação no nível de usuário.

Como podemos entender isso? Para qualquer sistema de arquivos ext2/3/4, comece pelo inode de número 2 – o *inode raiz*. A partir da tabela de inodes da figura 4.5, você pode ver que esse é um inode de diretório (dir), portanto você pode seguir a seta para o pool de dados, em que você verá o conteúdo do diretório-raiz: duas entradas de nomes *dir_1* e *dir_2* correspondentes aos inodes 12 e 7633, respectivamente. Para explorar essas entradas, retorne à tabela de inodes e dê uma olhada em qualquer um desses inodes.

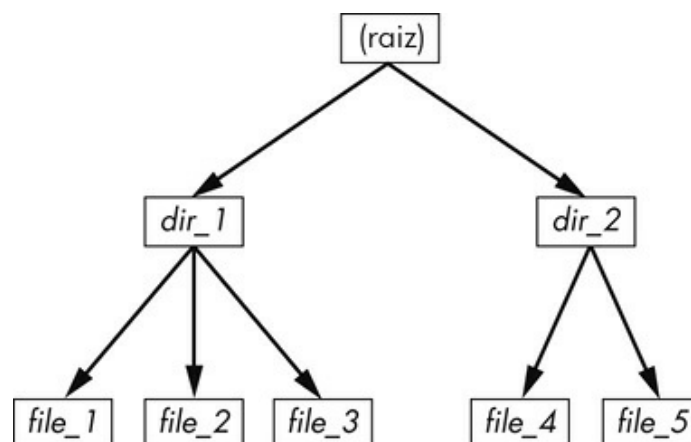


Figura 4.4 – Representação de um sistema de arquivos no nível de usuário.

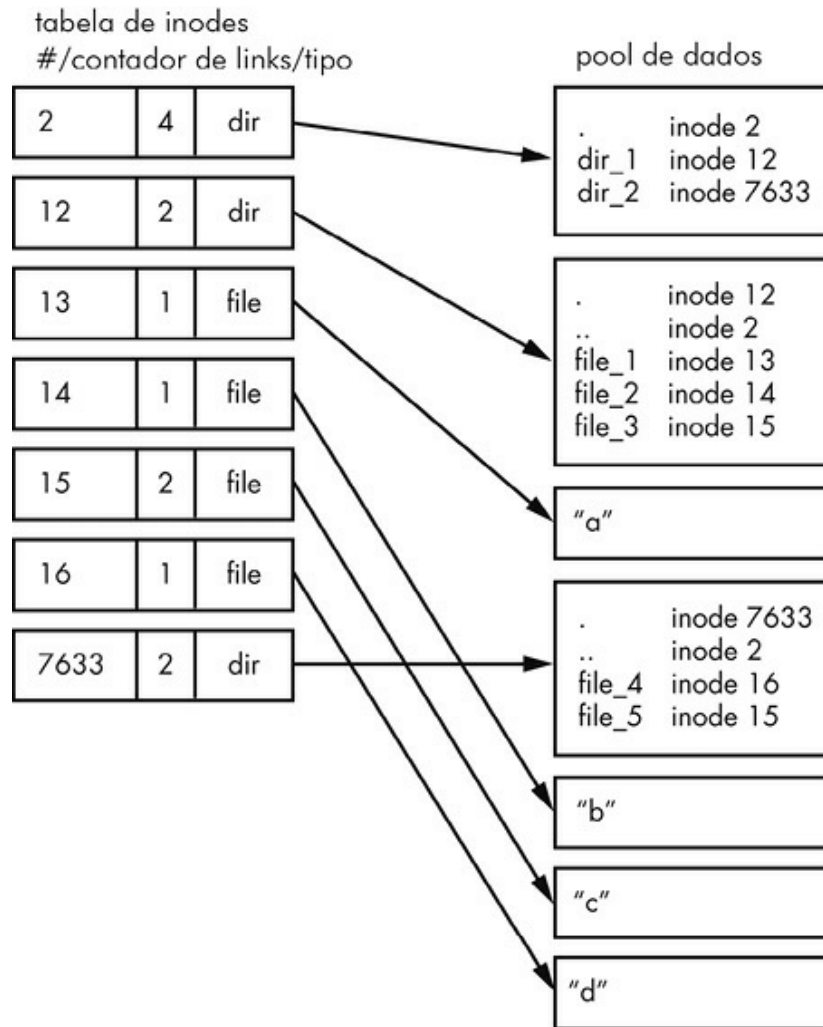


Figura 4.5 – Estrutura de inodes do sistema de arquivos mostrado na figura 4.4.

Para analisar *dir_1/file_2* nesse sistema de arquivos, o kernel faz o seguinte:

1. Determina os componentes do path: um diretório chamado *dir_1*, seguido de um componente chamado *file_2*.
2. Segue o inode da raiz para os dados de seu diretório.
3. Encontra o nome *dir_1* nos dados de diretório do inode 2, que aponta para o inode de número 12.
4. Procura o inode 12 na tabela de inodes e descobre que é um inode de diretório.
5. Segue o link de dados do inode 12 para obter as informações de diretório (a segunda caixa mais para baixo no pool de dados).
6. Localiza o segundo componente do path (*file_2*) nos dados de diretório do inode 12. Essa entrada aponta para o inode de número 14.
7. Procura o inode 14 na tabela de diretório. Esse é um inode de arquivo.

Nesse ponto, o kernel conhece as propriedades do arquivo e pode abri-lo seguindo o

link de dados do inode 14.

Esse sistema em que inodes apontam para estruturas de dados de diretório e estruturas de dados de diretório apontam para inodes permite criar a hierarquia de sistemas de arquivos com a qual você está acostumado. Além disso, observe que os inodes de diretório contêm entradas para `.` (o diretório corrente) e para `..` (o diretório-pai, exceto para o diretório-raiz). Isso facilita obter um ponto de referência e navegar de volta na estrutura de diretórios.

4.5.1 Visualizando detalhes de inodes

Para visualizar os números dos inodes de qualquer diretório, utilize o comando `ls -li`. Eis o que você obterá para a raiz nesse exemplo. (Para informações mais detalhadas de inode, utilize o comando `stat`.)

```
$ ls -li
12 dir_1 7633 dir_2
```

Provavelmente, você deve estar se perguntando sobre o contador de links. Você já viu o *contador de links* na saída do comando `ls -li` comum, porém é provável que o tenha ignorado. Como o contador de links se relaciona com os arquivos da figura 4.5, em particular, com o *file_5* com *hard link*? O campo de contador de links corresponde à quantidade total de entradas de diretório (em todos os diretórios) que aponta para um inode. A maioria dos arquivos tem um contador de links igual a 1, pois ocorre somente uma vez nas entradas de diretório. Isso é esperado: na maioria das vezes, ao criar um arquivo, você criará uma nova entrada de diretório e um novo inode para acompanhá-la. Entretanto o inode 15 ocorre duas vezes: inicialmente, ele é criado como *dir_1/file_3* e, em seguida, é ligado como *dir_2/file_5*. Um hard link é apenas uma entrada criada manualmente em um diretório para um inode que já existe. O comando `ln` (sem a opção `-s`) permite criar links novos manualmente.

É por esse motivo que, às vezes, a remoção de um arquivo é chamada de *unlinking* (remoção de link). Se você executar `rm dir_1/file_2`, o kernel procurará uma entrada chamada *file_2* nas entradas de diretório do inode 12. Ao descobrir que *file_2* corresponde ao inode 14, o kernel removerá a entrada de diretório e então subtrairá 1 do contador de links do inode 14. Como resultado, o contador de links do inode 14 será igual a 0 e o kernel saberá que não há mais nenhum nome fazendo a ligação com o inode. Sendo assim, ele poderá apagar o inode e qualquer dado associado a ele.

No entanto, se você executar `rm dir_1/file_3`, o resultado final será o contador de links do inode 15 ir de 2 para 1 (porque *dir_2/file_5* continua apontando para lá) e o kernel saberá que não deve remover o inode.

Os contadores de link funcionam de modo muito parecido para diretórios. Observe que o contador de links do inode 12 é 2 porque há dois links de inode ali: um para *dir_1* nas entradas de diretório para o inode 2 e o segundo é uma referência a si mesmo (.) nas entradas de seu próprio diretório. Se você criar um novo diretório *dir_1/dir_3*, o contador de links para o inode 12 irá para 3 porque o novo diretório incluirá uma entrada para o pai (..) que fará o link de volta para o inode 12, assim como o link pai do inode 12 aponta para o inode 2.

Há uma pequena exceção. O inode 2 da raiz tem um contador de links igual a 4. Porém a figura 4.5 mostra somente três links nas entradas de diretório. O “quarto” link está no superbloco do sistema de arquivos porque o superbloco informa o local em que o inode da raiz se encontra.

Não tenha medo de fazer experimentos em seu sistema. Criar uma estrutura de diretórios e, em seguida, usar `ls -i` ou `stat` para percorrer as partes é uma ação inofensiva. Não é preciso ser root (a menos que você monte e crie um novo sistema de arquivos).

Entretanto ainda há uma peça faltando: ao alocar blocos do pool de dados para um novo arquivo, como o sistema de arquivos sabe quais blocos estão em uso e quais estão disponíveis? Uma das maneiras básicas consiste no uso de uma estrutura de dados adicional de gerenciamento chamada *bitmap de blocos*. Nesse esquema, o sistema de arquivos reserva uma série de bytes, em que cada bit corresponde a um bloco no pool de dados. Um valor igual a 0 significa que o bloco está livre e 1 significa que está em uso. Desse modo, alocar e desalocar blocos é uma questão de virar os bits.

Os problemas em um sistema de arquivos surgem quando os dados da tabela de inodes não correspondem aos dados de alocação de blocos ou quando os contadores de link estão incorretos; isso pode acontecer quando um sistema não é desligado de forma apropriada. Sendo assim, ao efetuar a verificação de um sistema de arquivos, conforme descrito na seção 4.2.11, o programa `fsck` percorrerá a tabela de inodes e a estrutura de diretórios para gerar novos contadores de links e um novo mapa de alocação de blocos (por exemplo, o bitmap de blocos) e, em seguida, irá comparar os dados recém-gerados com o sistema de arquivos em disco. Se houver discrepâncias, `fsck` deverá corrigir os contadores de link e determinar o que fazer com qualquer inode e/ou dados que não aparecerem quando a estrutura de diretórios for percorrida. A maioria dos programas `fsck` transforma esses “órfãos” em novos arquivos no diretório *lost+found* do sistema de arquivos.

4.5.2 Trabalhando com sistemas de arquivos no espaço de usuário

Ao trabalhar com arquivos e diretórios no espaço de usuário, você não deverá se

preocupar muito com a implementação subjacente. Espera-se que você vá acessar o conteúdo dos arquivos e dos diretórios de um sistema de arquivos montado por meio de chamadas de sistema do kernel. Curiosamente, porém, você terá acesso a determinadas informações do sistema de arquivos que não parecem se encaixar no espaço de usuário – em particular, a chamada de sistema `stat()` retorna os números de inode e os contadores de links.

Quando não estiver fazendo manutenção em um sistema de arquivos, você deverá se preocupar com números de inode e com contadores de links? Em geral, não. Essas informações são acessíveis a programas em modo usuário principalmente por questões de compatibilidade com versões anteriores. Além do mais, nem todos os sistemas de arquivos disponíveis no Linux têm essas informações internas do sistema de arquivos. A camada de interface *VFS* (Virtual File System, ou Sistema de arquivos virtual) garante que as chamadas de sistema sempre retornem números de inode e contadores de link, porém esses números não necessariamente significam algo.

Você poderá não ser capaz de realizar operações tradicionais de sistemas de arquivos Unix em sistemas de arquivos não tradicionais. Por exemplo, não será possível usar `ln` para criar um hard link em um sistema de arquivos VFAT montado porque a estrutura da entradas de diretório é totalmente diferente.

Felizmente, as chamadas de sistema disponíveis no espaço de usuário em sistemas Unix/Linux oferecem abstração suficiente para um acesso sem dificuldades aos arquivos – não é preciso saber nada sobre a implementação subjacente para acessar os arquivos. Além do mais, os nomes de arquivo são flexíveis quanto ao formato e há suporte para nomes que misturem letras maiúsculas e minúsculas, facilitando o suporte a outros sistemas de arquivos de estilo hierárquico.

Lembre-se de que um suporte a sistemas de arquivos específicos não precisa estar necessariamente no kernel. Em sistemas de arquivos no espaço de usuário, o kernel deve atuar somente como um condutor para chamadas de sistema.

4.5.3 A evolução dos sistemas de arquivos

Como você pode ver, até o sistema de arquivos mais simples descrito tem vários componentes diferentes a serem mantidos. Ao mesmo tempo, as demandas sobre os sistemas de arquivos aumentam continuamente com novas tarefas, tecnologias e capacidades de armazenamento. O desempenho de hoje, a integridade dos dados e os requisitos de segurança estão além do que é oferecido pelas implementações de sistemas de arquivos mais antigos e, desse modo, a tecnologia dos sistemas de arquivos está em constante mudança. Já mencionamos o Btrfs como exemplo de um sistema de

arquivos da próxima geração (veja a seção 4.2.1).

Um exemplo de como os sistemas de arquivos estão mudando está no fato de que os novos sistemas usam estruturas de dados diferentes para representar diretórios e nomes de arquivo, em vez de usar os inodes de diretório descritos aqui. Eles fazem referência a blocos de dados de modo diferente. Além do mais, os sistemas de arquivos otimizados para SSDs continuam evoluindo. Mudanças contínuas no desenvolvimento de sistemas de arquivos é a norma, porém tenha em mente que a evolução desses sistemas não altera o seu propósito.

CAPÍTULO 5

Como o kernel do Linux inicializa

Você já conhece a estrutura física e lógica de um sistema Linux, o que é o kernel e como trabalhar com processos. Este capítulo mostrará como o kernel inicia, ou seja, efetua o boot. Em outras palavras, você aprenderá como o kernel se move para a memória até o ponto em que o primeiro processo de usuário é iniciado.

Uma visão simplificada do processo de boot tem o seguinte aspecto:

1. O BIOS do computador ou o firmware de boot é carregado e executa um boot loader.
2. O boot loader encontra a imagem do kernel no disco, carrega-a para a memória e a inicia.
3. O kernel inicializa os dispositivos e seus drivers.
4. O kernel monta o sistema de arquivos-raiz.
5. O kernel inicia um programa chamado *init* com um ID de processo igual a 1. Esse ponto é o *início do espaço de usuário*.
6. O *init* coloca o restante dos processos do sistema em ação.
7. Em algum momento, o *init* inicia um processo que permite que você faça login, normalmente no final ou próximo ao final do boot.

Este capítulo discute os quatro primeiros estágios, focando no kernel e nos boot loaders. O capítulo 6 prossegue com o início do espaço de usuário.

Sua capacidade de identificar cada estágio do processo de boot mostrará ter um valor inestimável na correção de problemas de boot e na compreensão do sistema como um todo. Entretanto o comportamento-padrão em muitas distribuições Linux geralmente faz com que seja difícil, se não impossível, identificar os primeiros estágios de boot à medida que ocorrem, portanto é provável que você vá poder dar uma boa olhada somente depois que essas fases tiverem sido concluídas e você tiver feito login.

5.1 Mensagens de inicialização

Sistemas Unix tradicionais geram várias mensagens de diagnóstico na inicialização, que

informam sobre o processo de boot. As mensagens inicialmente são provenientes do kernel e depois dos processos e dos procedimentos de inicialização começados pelo `init`. No entanto essas mensagens não são elegantes nem consistentes e, em alguns casos, não são nem mesmo muito informativas. A maioria das distribuições Linux atuais dão o melhor de si para ocultá-las com splash screens (telas de apresentação), fillers e opções de boot. Além do mais, as melhorias no hardware fazem o kernel iniciar muito mais rapidamente do que ocorria antigamente; as mensagens aparecem e desaparecem tão rapidamente que pode ser difícil ver o que está acontecendo.

Há duas maneiras de visualizar as mensagens de boot e de diagnóstico em tempo de execução do kernel. Você pode:

- Dar uma olhada no arquivo de log de sistema do kernel. Geralmente, esse arquivo pode ser encontrado em `/var/log/kern.log`, porém, de acordo com o modo como seu sistema estiver configurado, ele também poderá estar junto de vários outros logs de sistema em `/var/log/messages` ou em outro local.
- Utilizar o comando `dmesg`, mas não se esqueça de fazer pipe da saída para `less`, pois haverá muito mais que uma tela de informações. O comando `dmesg` usa o buffer circular do kernel, que tem tamanho limitado, porém a maioria dos kernels mais novos tem um buffer grande o suficiente para armazenar mensagens de boot durante um longo período de tempo.

A seguir, temos um exemplo do que você pode esperar ver em comando `dmesg`:

```
$ dmesg
[ 0.000000] Initializing cgroup subsys cpu
[ 0.000000] Linux version 3.2.0-67-generic-pae (buildd@toyol) (gcc version 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5))
) #101-Ubuntu SMP Tue Jul 15 18:04:54 UTC 2014 (Ubuntu 3.2.0-67.101-generic-pae 3.2.60)
[ 0.000000] KERNEL supported cpus:
--trecho omitido--
[ 2.986148] sr0: scsi3-mmc drive: 24x/8x writer dvd-ram cd/rw xa/form2 cdda tray
[ 2.986153] cdrom: Uniform CD-ROM driver Revision: 3.20
[ 2.986316] sr 1:0:0:0: Attached scsi CD-ROM sr0
[ 2.986416] sr 1:0:0:0: Attached scsi generic sg1 type 5
[ 3.007862] sda: sda1 sda2 < sda5 >
[ 3.008658] sd 0:0:0:0: [sda] Attached SCSI disk
--trecho omitido--
```

Após o kernel ter iniciado, o procedimento de inicialização do espaço de usuário normalmente gera mensagens. Essas mensagens provavelmente serão mais difíceis de serem visualizadas e analisadas porque, na maioria dos sistemas, você não as encontrará em um único arquivo de log. Os scripts de inicialização geralmente exibem

as mensagens no console, e elas são apagadas depois que o processo de boot é concluído. Contudo isso normalmente não é um problema, pois cada script geralmente escreve em seu próprio log. Algumas versões de init, como Upstart e systemd, podem capturar mensagens de diagnóstico da inicialização e em tempo de execução, que normalmente seriam enviadas ao console.

5.2 Inicialização do kernel e opções de boot

O kernel do Linux é inicializado nesta ordem geral:

1. Inspeção de CPU
2. Inspeção de memória
3. Descoberta de barramento de dispositivos
4. Descoberta de dispositivos
5. Configuração dos subsistemas auxiliares do kernel (rede e outros)
6. Montagem do sistema de arquivos-raiz
7. Inicialização do espaço de usuário

Os primeiros passos não são muito dignos de nota, porém, quando o kernel chega aos dispositivos, surge uma questão de dependência. Por exemplo, os device drivers de disco podem depender de suporte de barramento e de suporte ao subsistema SCSI.

Mais adiante no processo de inicialização, o kernel deve montar um sistema de arquivos raiz antes de iniciar o init. Em geral, você não precisará se preocupar com nada disso, exceto com o fato de que alguns componentes necessários poderão ser módulos carregáveis do kernel em vez de fazerem parte do kernel principal. Em alguns computadores, poderá ser necessário carregar esses módulos do kernel antes de o verdadeiro sistema de arquivos-raiz ser montado. Discutiremos esse problema e as soluções alternativas de sistema de arquivos inicial em RAM na seção 6.8.

Na época desta publicação, o kernel não emitia mensagens específicas quando estava prestes a iniciar o primeiro processo de usuário. Entretanto as mensagens de gerenciamento de memória a seguir são uma boa indicação de que a passagem para o espaço de usuário está para ocorrer, pois é nesse ponto que o kernel protege sua própria memória dos processos do espaço de usuário:

```
Freeing unused kernel memory: 740k freed
Write protecting the kernel text: 5820k
Write protecting the kernel read-only data: 2376k
NX-protecting the kernel data: 4420k
```

Você também poderá ver uma mensagem sobre o sistema de arquivos-raiz sendo montado nesse ponto.

 **Observação:** sinta-se à vontade para avançar para o capítulo 6 e conhecer informações específicas sobre a inicialização do espaço de usuário e o programa `init` executado pelo kernel como seu primeiro processo. O restante deste capítulo detalha o modo como o kernel inicia.

5.3 Parâmetros do kernel

Ao executar o kernel do Linux, o boot loader passa um conjunto de *parâmetros do kernel*, baseados em texto, que informam como o kernel deve iniciar. Os parâmetros especificam vários tipos diferentes de comportamento, por exemplo, a quantidade de saída referente a diagnósticos que o kernel deve gerar e opções específicas de device drivers.

Você pode visualizar os parâmetros de kernel do boot de seu sistema ao observar o arquivo `/proc/cmdline`:

```
$ cat /proc/cmdline
```

```
BOOT_IMAGE=/boot/vmlinuz-3.2.0-67-generic-pae root=UUID=70ccd6e7-6ae6-44f6-812c-51aab8036d29 ro quiet splash vt.handoff=7
```

Os parâmetros são flags de uma única palavra, por exemplo, `ro` e `quiet`, ou pares *chave=valor*, como em `vt.handoff=7`. Muitos dos parâmetros não são importantes, como a flag `splash` para exibir uma splash screen, porém um parâmetro crítico é `root`. Ele representa a localização do sistema de arquivos-raiz; sem ele, o kernel não poderá encontrar `init` e, desse modo, não poderá executar a inicialização do espaço de usuário.

O sistema de arquivos-raiz pode ser especificado como um arquivo de dispositivo, como neste exemplo:

```
root=/dev/sda1
```

Contudo, na maioria dos sistemas desktop modernos, um UUID é mais comum (veja a seção 4.2.4):

```
root=UUID=70ccd6e7-6ae6-44f6-812c-51aab8036d29
```

O parâmetro `ro` é normal; ele instrui o kernel a montar o sistema de arquivos-raiz em modo somente de leitura na inicialização do espaço de usuário. (O modo somente de leitura garante que `fsck` poderá verificar o sistema de arquivos-raiz de forma segura; após a verificação, o processo de inicialização remonta o sistema de arquivos-raiz em modo de leitura e de escrita.)

Ao encontrar um parâmetro que não seja compreendido, o kernel do Linux salvará o parâmetro. Mais tarde, ele o passará para `init`, quando a inicialização do espaço de

usuário for executada. Por exemplo, se você adicionar `-s` aos parâmetros do kernel, esse passará `-s` para o programa `init` para indicar que ele deverá ser iniciado em modo monousuário.

Vamos agora dar uma olhada no modo como os boot loaders iniciam o kernel.

5.4 Boot loaders

No início do processo de boot, antes de o kernel e o `init` iniciarem, um boot loader inicia o kernel. A tarefa de um boot loader parece ser simples: ele carrega o kernel na memória e, em seguida, inicia-o com um conjunto de parâmetros. Mas considere as perguntas que o boot loader deve responder:

- Onde está o kernel?
- Quais parâmetros de kernel devem ser passados quando o kernel for iniciado?

Segundo as respostas (típicas), o kernel e seus parâmetros geralmente estão em algum local do sistema de arquivos-raiz. Pode parecer que os parâmetros do kernel são fáceis de ser encontrados, exceto pelo fato de o kernel ainda não estar executando, portanto ele não poderá percorrer um sistema de arquivos para encontrar os arquivos necessários. Pior ainda, os device drivers do kernel, normalmente usados para acessar o disco, também não estão disponíveis. Pense nisso como uma espécie de problema do “ovo e da galinha”.

Vamos começar com o problema do driver. Nos PCs, os boot loaders usam o *BIOS* (Basic Input/Output System, ou Sistema básico de entrada/saída) ou o *UEFI* (Unified Extensible Firmware Interface, ou Interface unificada de firmware extensível) para acessar os discos. Quase todos os hardwares de disco têm firmwares que permitem ao BIOS acessar o hardware de armazenamento associado usando *LBA* (Linear Block Addressing, ou Endereçamento de bloco linear). Embora tenha um desempenho ruim, esse modo permite um acesso universal aos discos. Os boot loaders geralmente são os únicos programas que usam o BIOS para acesso a disco; o kernel utiliza seus próprios drivers de alto desempenho.

O problema do sistema de arquivos é mais complicado. A maioria dos boot loaders modernos consegue ler tabelas de partição e tem suporte embutido para acessos somente de leitura aos sistemas de arquivos. Desse modo, eles podem encontrar e ler arquivos. Essa capacidade facilita bastante configurar dinamicamente e aperfeiçoar o boot loader. Os boot loaders do Linux nem sempre tiveram essa capacidade; sem ela, configurar o boot loader era mais difícil.

5.4.1 Tarefas do boot loader

As principais funcionalidades de um boot loader do Linux incluem as capacidades de fazer o seguinte:

- Selecionar entre vários kernels.
- Alternar entre conjuntos de parâmetros de kernel.
- Permitir ao usuário sobrescrever manualmente e editar nomes de imagens e parâmetros do kernel (por exemplo, para entrar em modo monousuário).
- Oferecer suporte para inicialização de outros sistemas operacionais.

Os boot loaders se tornaram consideravelmente mais sofisticados desde o surgimento do kernel do Linux, com recursos como histórico e sistemas de menu, porém a necessidade básica sempre foi ter flexibilidade na seleção da imagem do kernel e dos parâmetros. Um fenômeno interessante é o fato de que determinadas necessidades foram reduzidas. Por exemplo, pelo fato de agora você poder realizar um boot de emergência ou de recuperação de modo parcial ou completo a partir de um dispositivo de armazenamento USB, provavelmente você não precisará se preocupar em inserir manualmente os parâmetros de kernel ou ativar o modo monousuário. Porém os boot loaders modernos jamais ofereceram tanta capacidade quanto atualmente, o que pode particularmente vir a calhar se você estiver criando kernels personalizados ou se quiser simplesmente ajustar os parâmetros.

5.4.2 Visão geral dos boot loaders

Eis os principais boot loaders que você poderá encontrar, em ordem de popularidade:

- GRUB – um padrão quase universal em sistemas Linux.
- LILO – um dos primeiros boot loaders de Linux. O ELILO é uma versão UEFI.
- SYSLINUX – pode ser configurado para executar a partir de vários tipos diferentes de sistemas de arquivos.
- LOADLIN – inicializa um kernel a partir do MS-DOS.
- efilinux – um boot loader UEFI cujo propósito é servir como modelo e referência para outros boot loaders UEFI.
- coreboot (anteriormente, o LinuxBIOS) – um substituto de alto desempenho para o PC BIOS que pode incluir um kernel.
- Linux Kernel EFISTUB – um plugin de kernel para carregar o kernel diretamente do ESP (EFI/UEFI System Partition) encontrado em sistemas recentes.

Este livro lida exclusivamente com o GRUB. O raciocínio por trás do uso de outros boot loaders está relacionado ao fato de eles serem mais simples de configurar do que o GRUB ou serem mais rápidos.

Para fornecer um nome e parâmetros de kernel, inicialmente é preciso saber como obter um prompt de boot. Infelizmente, isso às vezes pode ser difícil de descobrir porque as distribuições Linux personalizam o comportamento e a aparência do boot loader a seu bel-prazer.

As próximas seções mostrarão como obter um prompt de boot para fornecer um nome e os parâmetros do kernel. Depois que se sentir à vontade com isso, você verá como configurar e como instalar um boot loader.

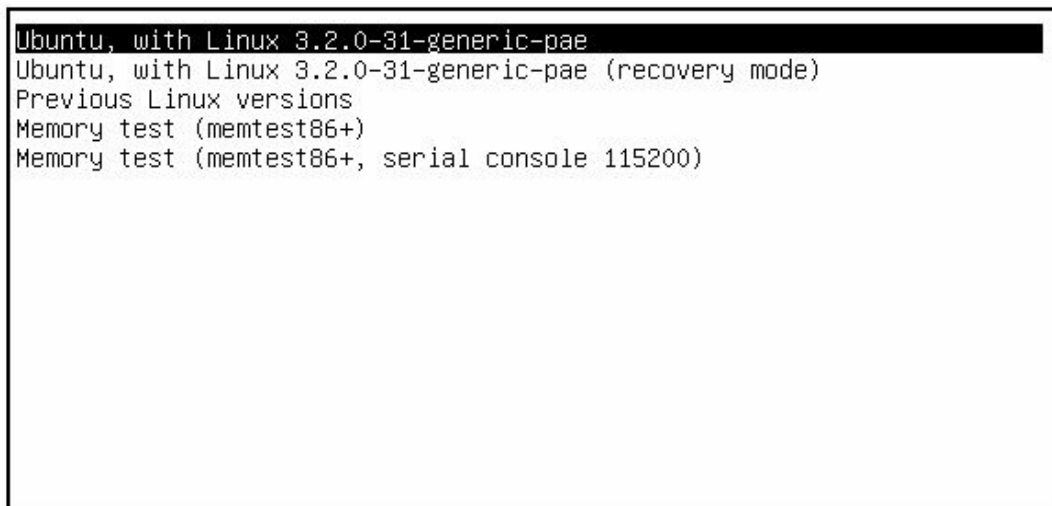
5.5 Introdução ao GRUB

GRUB quer dizer *Grand Unified Boot Loader*. Discutiremos o GRUB 2. Há também uma versão mais antiga atualmente chamada de GRUB Legacy, que, aos poucos, está caindo em desuso.

Um dos recursos mais importantes do GRUB é a navegação em sistemas de arquivos, que permite uma seleção muito mais fácil da imagem e da configuração do kernel. Uma das melhores maneiras de ver isso em ação e conhecer o GRUB em geral é dar uma olhada em seu menu. É fácil navegar pela sua interface, porém há uma boa chance de que você ainda não a tenha visto. As distribuições Linux geralmente fazem o máximo para ocultar o boot loader de você.

Para acessar o menu do GRUB, pressione e segure a tecla Shift quando a tela de BIOS ou de inicialização do firmware aparecer pela primeira vez. Caso contrário, a configuração do boot loader não fará uma pausa antes de carregar o kernel. A figura 5.1 mostra o menu do GRUB. Tecle Esc para desabilitar temporariamente o timeout automático de boot depois que o menu do GRUB aparecer.

GNU GRUB version 1.99-21ubuntu3.1



Use the ↑ and ↓ keys to select which entry is highlighted.
Press enter to boot the selected OS, 'e' to edit the commands
before booting or 'c' for a command-line.

Figura 5.1 – Menu do GRUB.

Tente fazer o seguinte para explorar o boot loader:

1. Reinicie ou ligue o seu sistema Linux.
2. Segure a tecla Shift durante o autoteste de BIOS/Firmware e/ou a splash screen para acessar o menu do GRUB.
3. Tecle e para visualizar os comandos de configuração do boot loader para a opção de boot default. Você deverá ver algo semelhante ao que está sendo mostrado na figura 5.2.

Essa tela nos informa que, para essa configuração, a raiz está definida com um UUID, a imagem do kernel é `/boot/vmlinuz-3.2.0-31-generic-pae` e os parâmetros de kernel incluem `ro`, `quiet` e `splash`. O sistema de arquivos inicial em RAM é `/boot/initrd.img-3.2.0-31-generic-pae`. No entanto, se você nunca viu esse tipo de configuração antes, poderá achá-la, de certo modo, confusa. Por que há várias referências a `root` e por que elas são diferentes? Por que `insmod` está aqui? Esse não é um recurso do kernel do Linux, normalmente executado por `udev`?

```
setparams 'Ubuntu, with Linux 3.2.0-31-generic-pae'

recordfail
gfxmode $linux_gfx_mode
insmod gzio
insmod part_msdos
insmod ext2
set root='(hd0,msdos1)'
search --no-floppy --fs-uuid --set=root 4898e145-b064-45bd-b7b4-7326\
b00273b7
linux /boot/vmlinuz-3.2.0-31-generic-pae root=UUID=4898e145-b064-45b\
d-b7b4-7326b00273b7 ro quiet splash $vt_handoff
initrd /boot/initrd.img-3.2.0-31-generic-pae
```

Minimum Emacs-like screen editing is supported. TAB lists completions. Press Ctrl-x or F10 to boot, Ctrl-c or F2 for a command-line or ESC to discard edits and return to the GRUB menu.

Figura 5.2 – Editor de configuração do GRUB.

A surpresa final é garantida pelo fato de o GRUB não *usar* realmente o kernel do Linux – ele o *inicia*. A configuração que você vê é totalmente constituída de comandos internos do GRUB. O GRUB realmente é um mundo todo à parte.

A confusão origina-se do fato de o GRUB emprestar uma terminologia de várias fontes. O GRUB tem seu próprio “kernel” e seu próprio comando `insmod` para carregar módulos do GRUB de forma dinâmica, de modo totalmente independente do kernel do Linux. Muitos comandos do GRUB são semelhantes aos comandos do shell Unix; há até mesmo um comando `ls` para listar arquivos.

Porém a maior parte da confusão é proveniente do uso da palavra *root*. Para esclarecer, há uma regra simples a ser seguida quando você estiver procurando o sistema de arquivos-raiz de seu sistema: *somente* o parâmetro *root* do *kernel* será o sistema de arquivos-raiz quando o seu sistema for inicializado.

Na configuração do GRUB, o parâmetro do kernel está em algum ponto após o nome da imagem no comando `linux`. Todas as demais referências a *root* na configuração relacionam-se ao root do GRUB, que existe somente no GRUB. O “root” do GRUB corresponde ao sistema de arquivos no qual o GRUB procura os arquivos de imagem do kernel e do sistema de arquivos em RAM.

Na figura 5.2, o root do GRUB inicialmente é definido para um dispositivo específico do GRUB (`hd0,msdos1`). Então, no comando seguinte, o GRUB procura um UUID em particular em uma partição. Se esse UUID for encontrado, o root do GRUB será definido para essa partição.

Para concluir, o primeiro argumento do comando `linux (/boot/vmlinuz-...)` corresponde à localização do arquivo de imagem do kernel do Linux. O GRUB carrega esse arquivo a partir do root do GRUB. O comando `initrd` é semelhante e especifica o arquivo para o sistema de arquivos inicial em RAM.

Você pode editar essa configuração no GRUB; fazer isso geralmente é a maneira mais fácil de corrigir temporariamente um boot incorreto. Para corrigir permanentemente um problema de boot, será preciso alterar a configuração (veja a seção 5.5.2), mas, por enquanto, vamos avançar mais um passo e analisar alguns detalhes internos do GRUB com a interface de linha de comando.

5.5.1 Explorando dispositivos e partições com a linha de comando do GRUB

Como você pode ver na figura 5.2, o GRUB tem seu próprio esquema de endereçamento de dispositivos. Por exemplo, o primeiro disco rígido encontrado é `hd0`, seguido de `hd1` e assim por diante. Porém as atribuições de dispositivo estão sujeitas a mudanças. Felizmente, o GRUB pode pesquisar todas as partições em busca de um UUID para encontrar aquela em que está o kernel, como você acabou de ver no comando `search`.

Listando dispositivos

Para ter uma noção de como o GRUB referencia os dispositivos em seu sistema, acesse a linha de comando do GRUB ao teclar `C` no menu de boot ou no editor de configuração. O prompt do GRUB deverá ser obtido:

```
grub>
```

Você pode especificar aqui qualquer comando que você verá em uma configuração, mas, para começar, procure executar um comando de diagnóstico: `ls`. Sem argumentos, a saída será uma lista de dispositivos conhecidos pelo GRUB:

```
grub> ls
(hd0) (hd0,msdos1) (hd0,msdos5)
```

Nesse caso, há um dispositivo de disco principal representado por `(hd0)` e as partições `(hd0,msdos1)` e `(hd0,msdos5)`. O prefixo *msdos* nas partições informa que o disco contém uma tabela de partição MBR; o nome começaria com *gpt* para GPT. (Você encontrará combinações até mais elaboradas, com um terceiro identificador, em que um

mapa de rótulos de disco BSD reside em uma partição, porém, normalmente, não será necessário se preocupar com isso, a menos que você esteja executando vários sistemas operacionais em um computador.)

Para obter informações mais detalhadas, utilize `ls -l`. Esse comando pode ser particularmente útil, pois ele exibe qualquer UUID das partições no disco. Por exemplo:

```
grub> ls -l
Device hd0: Not a known filesystem - Total size 426743808 sectors
  Partition hd0,msdos1: Filesystem type ext2 – Last modification time
    2015-09-18 20:45:00 Friday, UUID 4898e145-b064-45bd-b7b4-7326b00273b7 -
Partition start at 2048 - Total size 424644608 sectors
  Partition hd0,msdos5: Not a known filesystem - Partition start at
    424648704 - Total size 2093056 sectors
```

Esse disco em particular tem um sistema de arquivos Linux ext2/3/4 na primeira partição MBR e uma assinatura de swap Linux na partição 5, que é uma configuração bem comum. [Você não pode dizer, porém, que (hd0,msdos5) é uma partição de swap a partir desse resultado.]

Navegação em arquivos

Vamos agora dar uma olhada nos recursos de navegação do sistema de arquivos do GRUB. Determine a raiz do GRUB por meio do comando `echo` (lembre-se de que é nesse local que o GRUB espera encontrar o kernel):

```
grub> echo $root
hd0,msdos1
```

Para usar o comando `ls` do GRUB a fim de listar os arquivos e os diretórios nessa raiz, você pode concatenar uma barra para a frente no final da partição:

```
grub> ls (hd0,msdos1)/
```

Porém é difícil lembrar e digitar a partição da raiz, portanto use a variável `root` para economizar tempo:

```
grub> ls ($root)/
```

A saída consiste de uma breve lista de nomes de arquivos e de diretórios no sistema de arquivos dessa partição, por exemplo, *etc/*, *bin/* e *dev/*. Você deverá perceber que essa é uma função totalmente diferente do `ls` do GRUB: anteriormente, você estava listando dispositivos, tabelas de partição e, talvez, algumas informações de cabeçalho do sistema de arquivos. Agora você está vendo o conteúdo propriamente dito dos sistemas de arquivo.

Podemos dar uma olhada mais detalhada nos arquivos e diretórios de uma partição de maneira semelhante. Por exemplo, para inspecionar o diretório */boot*, comece com o seguinte comando:

```
grub> ls ($root)/boot
```

👍 **Observação:** utilize as setas para cima e para baixo para percorrer o histórico de comandos do GRUB, e as setas para a esquerda e para a direita para editar a linha de comando atual. As teclas-padrão de leitura de linha (Ctrl-N, Ctrl-P e assim por diante) também funcionam.

Você também pode ver todas as variáveis do GRUB definidas no momento usando o comando `set`:

```
grub> set
?=0
color_highlight=black/white
color_normal=white/black
--trecho omitido--
prefix=(hd0,msdos1)/boot/grub
root=hd0,msdos1
```

Uma das variáveis mais importante é `$prefix` — o sistema de arquivos e o diretório em que o GRUB espera encontrar sua configuração e suporte auxiliar. Iremos explorar isso na próxima seção.

Depois que tiver terminado de usar a interface de linha de comando do GRUB, dê o comando `boot` para inicializar sua configuração atual ou simplesmente tecle `Esc` para retornar ao menu do GRUB. Qualquer que seja o caso, inicie o seu sistema; iremos explorar a configuração do GRUB, e isso será mais adequado quando você tiver todo o seu sistema disponível.

5.5.2 Configuração do GRUB

O diretório de configuração do GRUB contém o arquivo de configuração principal (*grub.cfg*) e vários módulos carregáveis, com sufixo *.mod*. (À medida que as versões de GRUB evoluem, esses módulos são transferidos para subdiretórios, como *i386-pc*.) O diretório normalmente é */boot/grub* ou */boot/grub2*. Não iremos modificar *grub.cfg* diretamente; em vez disso, usaremos o comando `grub-mkconfig` (ou `grub2-mkconfig` no Fedora).

Analisando o arquivo *grub.cfg*

Inicialmente, dê uma olhada rápida em *grub.cfg* para ver como o GRUB inicializa seu menu e as opções de kernel. Você verá que o arquivo *grub.cfg* é constituído de comandos do GRUB, que normalmente começam com vários passos de inicialização, seguidos de uma série de entradas de menu para diferentes configurações de kernel e de

boot. A inicialização não é complicada; é um conjunto de definições de funções e de comandos de configuração de vídeo como este:

```
if loadfont /usr/share/grub/unicode.pf2 ; then
    set gfxmode=auto
    load_video
    insmod gfxterm
--trecho omitido--
```

Mais adiante nesse arquivo, você deverá ver as configurações de boot disponíveis, cada qual começando com o comando `menuentry`. Você deverá ser capaz de ler e de entender esse exemplo com base no que você aprendeu na seção anterior:

```
menuentry 'Ubuntu, with Linux 3.2.0-34-generic-pae' --class ubuntu --class gnu-linux --class gnu
--class os {
    recordfail
    gfxmode $linux_gfx_mode
    insmod gzio
    insmod part_msdos
    insmod ext2
    set root='(hd0,msdos1)'
    search --no-floppy --fs-uuid --set=root 70ccd6e7-6ae6-44f6-812c-51aab8036d29
    linux /boot/vmlinuz-3.2.0-34-generic-pae root=UUID=70ccd6e7-6ae6-44f6-812c-51aab8036d29
    ro quiet splash $vt_handoff
    initrd /boot/initrd.img-3.2.0-34-generic-pae
}
```

Preste atenção nos comandos `submenu`. Se o seu arquivo *grub.cfg* contiver vários comandos `menuentry`, a maioria deles provavelmente estará encapsulada em um comando `submenu` para versões mais antigas do kernel para que o menu do GRUB não fique congestionado.

Gerando um novo arquivo de configuração

Se você quiser fazer alterações na configuração do GRUB, não edite seu arquivo *grub.cfg* diretamente, pois ele é gerado automaticamente e o sistema ocasionalmente o sobrescreve. Você irá adicionar sua nova configuração em outro lugar e, em seguida, irá executar `grub-mkconfig` para gerar a nova configuração.

Para ver como a geração da configuração funciona, dê uma olhada no início de *grub.cfg*. Deve haver linhas comentadas como esta:

```
### BEGIN /etc/grub.d/00_header ###
```

Ao inspecioná-las melhor, você perceberá que todo arquivo em */etc/grub.d* é um shell script que gera uma parte do arquivo *grub.cfg*. O próprio comando `grub-mkconfig` é um

shell script que executa tudo em */etc/grub.d*.

Tente executá-lo você mesmo como root. (Não se preocupe em sobrescrever a sua configuração atual. Esse comando sozinho simplesmente exibe a configuração na saída-padrão.)

```
# grub-mkconfig
```

E se você quiser inserir entradas de menu e outros comandos na configuração do GRUB? A resposta breve é que você deve colocar suas personalizações em um novo arquivo *custom.cfg* no diretório de configuração do GRUB, por exemplo, em */boot/grub/custom.cfg*.

A resposta longa é um pouco mais complicada. O diretório de configuração */etc/grub.d* oferece duas opções: *40_custom* e *41_custom*. A primeira opção, *40_custom*, é um script que você mesmo pode editar, mas, provavelmente, é a menos estável; um upgrade de pacote poderá destruir qualquer alteração que você fizer. O script *41_custom* é mais simples; é somente uma série de comandos que carrega *custom.cfg* quando o GRUB inicia. (Tenha em mente que, se você escolher essa segunda opção, suas alterações não aparecerão quando você gerar o seu arquivo de configuração.)

As duas opções para arquivos de configuração personalizados não são particularmente extensíveis. Você verá adições no diretório particular */etc/grub.d* de sua distribuição. Por exemplo, o Ubuntu adiciona opções de boot para testes de memória (*memtest86+*) à configuração.

Para escrever e instalar um arquivo de configuração GRUB recém-gerado, você poderá gravar a configuração em seu diretório do GRUB usando a opção *-o* de *grub-mkconfig*, desta maneira:

```
# grub-mkconfig -o /boot/grub/grub.cfg
```

Ou, se você for um usuário de Ubuntu, basta executar *install-grub*. Qualquer que seja o caso, faça backup de sua configuração antiga, certifique-se de estar instalando no diretório correto e assim por diante.

Agora entraremos em alguns dos detalhes mais técnicos do GRUB e dos boot loaders. Se você estiver cansado de ouvir a respeito dos boot loaders e do kernel, sintase à vontade para pular para o capítulo 6.

5.5.3 Instalação do GRUB

Instalar o GRUB é uma operação mais sofisticada do que configurá-lo. Felizmente, você não precisará se preocupar normalmente com a instalação porque a sua distribuição deverá cuidar disso. No entanto, se você estiver tentando duplicar ou

restaurar um disco de boot ou se estiver preparando a sua própria sequência de boot, pode ser que seja necessário instalá-lo por conta própria.

Antes de prosseguir, leia a seção 5.8.3 para ter uma ideia de como os PCs fazem boot e verifique se você está usando o boot MBR ou EFI. Em seguida, crie o conjunto de software do GRUB e defina o local em que estará o seu diretório do GRUB; o default é */boot/grub*. Pode ser que não seja preciso criar o GRUB caso a sua distribuição o fizer por você, mas, se for necessário criá-lo, consulte o capítulo 16 para saber como criar o software a partir do código-fonte. Certifique-se de que o alvo correto seja criado: ele é diferente para boot MBR ou UEFI (e há até mesmo diferenças entre EFI 32 bits e 64 bits).

Instalando o GRUB em seu sistema

A instalação do boot loader exige que você ou um instalador determine o seguinte:

- O diretório de GRUB alvo, como visto pelo seu sistema em execução no momento. Normalmente, é */boot/grub*, porém poderá ser diferente se você estiver instalando o GRUB em outro disco para ser usado em outro sistema.
- O dispositivo corrente do disco-alvo do GRUB.
- Para booting UEFI, o ponto de montagem atual da partição de boot do UEFI.

Lembre-se de que o GRUB é um sistema modular, mas, para carregar os módulos, ele deverá ler o sistema de arquivos que contém o diretório do GRUB. Sua tarefa é criar uma versão de GRUB capaz de ler esse sistema de arquivos para que o restante de sua configuração (*grub.cfg*) possa ser carregado, além de qualquer módulo necessário. No Linux, isso geralmente significa criar uma versão de GRUB com seu módulo *ext2.mod* previamente carregado. Depois que você tiver essa versão, tudo o que é necessário fazer é colocá-la na parte do disco passível de boot e inserir o restante dos arquivos necessários em */boot/grub*.

Felizmente, o GRUB vem com um utilitário chamado `grub-install` (não o confunda com o `install-grub` do Ubuntu), que realiza a maior parte do trabalho de instalação dos arquivos do GRUB e da configuração para você. Por exemplo, se o seu disco atual estiver em */dev/sda* e você quiser instalar o GRUB nesse disco com o seu diretório */boot/grub* atual, utilize este comando para instalar o GRUB no MBR:

```
# grub-install /dev/sda
```

 **AVISO:** instalar o GRUB de modo incorreto pode acarretar falha na sequência de inicialização de seu sistema, portanto não use esse comando levemente. Se você estiver preocupado, leia sobre como fazer backup de seu MBR com `dd`, faça backup de qualquer outro diretório GRUB instalado no momento e certifique-se de ter um plano de inicialização de emergência.

Instalando o GRUB em um dispositivo de armazenamento externo

Para instalar o GRUB em um dispositivo de armazenamento fora do sistema corrente, você deverá especificar manualmente o diretório do GRUB nesse dispositivo do modo como o seu sistema atual o vê no momento. Por exemplo, suponha que você tenha um dispositivo-alvo igual a `/dev/sdc` e que o sistema de arquivos-raiz/arquivos de boot desse dispositivo (por exemplo, `/dev/sdc1`) esteja montado em `/mnt` em seu sistema atual. Isso implica que, ao instalar o GRUB, seu sistema atual verá os arquivos do GRUB em `/mnt/boot/grub`. Ao executar `grub-install`, informe-lhe o local em que esses arquivos estarão da seguinte maneira:

```
# grub-install --boot-directory=/mnt/boot /dev/sdc
```

Instalando o GRUB com UEFI

A instalação UEFI supostamente deve ser mais simples, pois tudo o que você deve fazer é copiar o boot loader no lugar certo. Porém você também deverá “anunciar” o boot loader ao firmware com o comando `efibootmgr`. O comando `grub-install` executa isso se estiver disponível, portanto, teoricamente, tudo o que você deve fazer para efetuar a instalação em uma partição UEFI é o seguinte:

```
# grub-install --efi-directory=efi_dir --bootloader-id=nome
```

Nesse caso, *efi_dir* é o local em que o diretório UEFI aparece em seu sistema atual (normalmente é `/boot/efi/efi`, pois a partição UEFI em geral é montada em `/boot/efi`) e *nome* é um identificador para o boot loader, conforme descrito na seção 5.8.2.

Infelizmente, muitos problemas podem surgir ao instalar um boot loader UEFI. Por exemplo, se você estiver fazendo a instalação em um disco que, em algum momento, for usado em outro sistema, você deverá descobrir como anunciar esse boot loader ao firmware do novo sistema. E há diferenças no procedimento de instalação para mídias removíveis.

Porém um dos principais problemas é o boot seguro (secure boot) do UEFI.

5.6 Problemas com o boot seguro do UEFI

Um dos problemas mais recentes que afetam as instalações Linux é o recurso de boot seguro encontrado nos PCs mais novos. Quando está ativo, esse sistema no UEFI exige que os boot loaders sejam digitalmente assinados por uma autoridade confiável para serem executados. A Microsoft exigiu que os fornecedores que disponibilizem o Windows 8 utilizem o boot seguro. O resultado é que se você tentar instalar um boot loader não assinado (o que ocorre na maioria das distribuições Linux atuais), ele não

será carregado.

A maneira mais fácil de contornar esse problema para qualquer pessoa que não tiver nenhum interesse em Windows é desabilitar o boot seguro nas configurações EFI. Entretanto isso não funcionará de forma limpa em sistemas com boot dual e poderá não ser uma opção para todos os usuários. Por causa disso, as distribuições Linux estão oferecendo boot loaders assinados. Algumas soluções são apenas frontends para o GRUB, outras oferecem uma sequência de carga totalmente assinada (desde o boot loader até o kernel) e outras são boot loaders totalmente novos (alguns baseados no efilinux).

5.7 Carregando outros sistemas operacionais em cadeia

O UEFI faz com que seja relativamente fácil suportar a carga de outros sistemas operacionais porque você pode instalar vários boot loaders na partição EFI. No entanto, o estilo MBR mais antigo não suporta isso e, apesar de ter UEFI, você ainda poderá ter uma partição individual com um boot loader estilo MBR que você queira usar. Você pode fazer o GRUB carregar e executar um boot loader diferente em uma partição específica de seu disco ao efetuar uma *carga em cadeia* (chainloading).

Para efetuar uma carga em cadeia, crie uma nova entrada de menu em sua configuração do GRUB (usando um dos métodos da página 139). Aqui está um exemplo para uma instalação Windows na terceira partição de um disco:

```
menuentry "Windows" {  
    insmod chain  
    insmod ntfs  
    set root=(hd0,3)  
    chainloader +1  
}
```

A opção `+1` de `chainloader` informa-lhe para carregar o que estiver no primeiro setor de uma partição. Você também pode obter isso ao carregar diretamente um arquivo usando uma linha como esta para carregar o loader *io.sys* de MS-DOS:

```
menuentry "DOS" {  
    insmod chain  
    insmod fat  
    set root=(hd0,3)  
    chainloader /io.sys  
}
```

5.8 Detalhes dos boot loaders

Agora daremos uma olhada rápida em alguns detalhes internos dos boot loaders. Sinta-se à vontade para pular para o próximo capítulo caso esse material não seja de seu interesse.

Para entender como os boot loaders como o GRUB funcionam, vamos inicialmente analisar como um PC faz boot ao ser ligado. Devido às repetidas inadequações dos sistemas de boot tradicionais dos PCs, há diversas variantes, porém há dois esquemas principais: MBR e UEFI.

5.8.1 Boot MBR

Além das informações sobre partição descritas na seção 4.1, o *MBR* (Master Boot Record) inclui uma pequena área (441 bytes) que o PC BIOS carrega e executa depois de seu *POST* (Power-On Self-Test). Infelizmente, é uma área muito pequena para armazenar praticamente qualquer boot loader, portanto um espaço adicional é necessário, resultando no que, às vezes, é chamado de *multi-stage boot loader* (boot loader de múltiplos estágios). Nesse caso, a parte inicial do código no MBR não faz nada além de carregar o restante do código do boot loader. As partes restantes do boot loader normalmente são carregadas no espaço entre o MBR e a primeira partição do disco.

É claro que isso não é extremamente seguro, pois qualquer um poderá sobrescrever o código nesse local, porém a maioria dos boot loaders faz isso, incluindo a maior parte das instalações GRUB. Além do mais, esse esquema não funcionará com um disco particionado com GPT usando o BIOS para fazer o boot porque as informações da tabela GPT estão em uma área após o MBR. (O GPT não interfere no MBR tradicional por questão de compatibilidade com versões anteriores.)

A solução alternativa para o GPT consiste em criar uma pequena partição chamada *partição de boot do BIOS* com um UUID especial para proporcionar um local em que o código completo do boot loader seja armazenado. Porém o GPT normalmente é usado com UEFI, e não com o BIOS tradicional, o que nos leva ao esquema de boot UEFI.

5.8.2 Boot UEFI

Os fabricantes de PCs e as empresas de software perceberam que o PC BIOS tradicional é severamente limitado e, por isso, decidiram desenvolver um substituto chamado *EFI* (Extensible Firmware Interface, ou Interface de firmware extensível). O EFI levou um tempo para ser adotado na maioria dos PCs, porém, atualmente, é

bastante comum. O padrão atual é o *UEFI* (Unified EFI, ou EFI unificado), que inclui recursos como um shell embutido e a capacidade de ler tabelas de partição e navegar por sistemas de arquivos. O esquema de particionamento GPT faz parte do padrão UEFI.

O booting é radicalmente diferente em sistemas UEFI e, em sua maior parte, é muito mais fácil de entender. Em vez de ter um código de boot executável armazenado fora de um sistema de arquivos, há sempre um sistema de arquivos especial chamado *ESP* (EFI System Partition), que contém um diretório chamado *efi*. Cada boot loader tem seu próprio identificador e um subdiretório correspondente, como *efi/microsoft*, *efi/apple* ou *efi/grub*. Um arquivo de boot loader tem uma extensão *.efi* e fica em um desses subdiretórios, juntamente com outros arquivos de suporte.

 **Observação:** o ESP difere da partição de boot do BIOS descrita na seção 5.8.1 e tem um UUID diferente.

Mas há um porém: você não pode simplesmente colocar um código antigo de boot loader no ESP porque esse código foi escrito para a interface BIOS. Em vez disso, você deve fornecer um boot loader escrito para UEFI. Por exemplo, ao usar o GRUB, você deverá instalar a versão UEFI do GRUB em vez de usar a versão BIOS. Além disso, será necessário “anunciar” novos boot loaders ao firmware.

E, como mencionamos na seção 5.6, temos a questão do “boot seguro”.

5.8.3 Como o GRUB funciona

Vamos concluir nossa discussão sobre o GRUB dando uma olhada em como ele faz o seu trabalho

1. O PC BIOS ou o firmware inicializam o hardware e pesquisam seus dispositivos de armazenamento de ordem de boot em busca do código de boot.
2. Ao encontrar o código de boot, o BIOS/firmware o carrega e o executa. É nesse ponto que o GRUB inicia.
3. O núcleo (core) do GRUB é carregado.
4. O núcleo é inicializado. Nesse ponto, o GRUB pode acessar discos e sistemas de arquivo.
5. O GRUB identifica sua partição de boot e carrega uma configuração nesse local.
6. O GRUB dá uma chance ao usuário para alterar a configuração.
7. Após um timeout ou uma ação do usuário, o GRUB executa a configuração (a sequência de comandos descrita na seção 5.5.2).

8. Durante a execução da configuração, o GRUB poderá carregar códigos adicionais (*módulos*) na partição de boot.
9. O GRUB executa um comando `boot` para carregar e executar o kernel, conforme especificado pelo comando `linux` da configuração.

Os passos 3 e 4 da sequência anterior, em que o núcleo do GRUB é carregado, podem ser complicados devido às repetidas inadequações dos sistemas tradicionais de boot de PC. A principal pergunta é “em que local *está* o núcleo do GRUB?”. Há três possibilidades básicas:

- parcialmente inserido entre o MBR e o início da primeira partição;
- em uma partição normal;
- em uma partição de boot especial: uma partição de boot GPT, no ESP (EFI System Partition) ou em outro local.

Em todos os casos, exceto naquele em que temos um ESP, o PC BIOS carregará 512 bytes do MBR, e é aí que o GRUB inicia. Essa pequena parte (derivada de *boot.img* no diretório do GRUB) ainda não é o núcleo, mas contém o local de início do núcleo e o carrega a partir desse ponto.

Entretanto, se você tiver um ESP, o núcleo do GRUB estará ali na forma de um arquivo. O firmware pode navegar pelo ESP e executar diretamente o núcleo do GRUB ou o carregador de qualquer outro sistema operacional localizado aí.

Apesar disso, na maioria dos sistemas, esse não é o quadro completo. O boot loader também poderá ter de carregar uma imagem inicial de um sistema de arquivos em RAM para a memória antes de carregar e de executar o kernel. É isso que o parâmetro de configuração `initrd` da seção 6.8 especifica. Porém, antes de conhecer o sistema de arquivos inicial em RAM, você deverá aprender sobre a inicialização do espaço de usuário – é a partir daí que o próximo capítulo começa.

CAPÍTULO 6

Como o espaço de usuário inicia

O ponto em que o kernel inicia o seu primeiro processo do espaço de usuário – o `init` – é significativo não só porque é o momento em que a memória e a CPU finalmente estão prontas para a operação normal do sistema, mas também porque é quando você pode ver como o restante do sistema é construído como um todo. Antes desse ponto, o kernel executa um caminho de execução bem controlado, definido por um número relativamente pequeno de desenvolvedores de software. O espaço de usuário é muito mais modular. É bem mais fácil ver o que ocorre na inicialização e na operação do espaço de usuário. Para os mais aventureiros, é também relativamente mais fácil modificar a inicialização do espaço de usuário, pois fazer isso não exige programação de baixo nível.

O espaço de usuário, de modo geral, inicia nesta ordem:

1. `init`
2. Serviços essenciais de baixo nível, como `udev` e `syslogd`
3. Configuração de rede
4. Serviços de nível médio e alto (`cron`, impressão e assim por diante)
5. Prompts de login, GUIs e outras aplicações de alto nível

6.1 Introdução ao `init`

O programa `init` é um programa do espaço de usuário como qualquer outro do sistema Linux, e você o encontrará em `/sbin`, juntamente com vários outros binários do sistema. Seu propósito principal é iniciar e finalizar os processos de serviços essenciais do sistema, porém versões mais recentes têm mais responsabilidades.

Há três principais implementações do `init` nas distribuições Linux:

- System V `init` – um `init` sequenciado tradicional (Sys V, normalmente pronuncia-se em inglês como “sys-five”). O Red Hat Enterprise Linux e várias outras distribuições utilizam essa versão.
- `systemd` – o padrão emergente para o `init`. Muitas distribuições passaram a usar o

systemd, e a maioria das que ainda não o fizeram planejam usá-lo.

- Upstart – o init em instalações Ubuntu. Entretanto, na época desta publicação, o Ubuntu também tinha planos para migrar para o systemd.

Há também várias outras versões de init, especialmente em plataformas embarcadas. Por exemplo, o Android tem o seu próprio init. Os BSDs também têm sua versão de init, porém é improvável que você vá vê-las em um computador Linux moderno. (Algumas distribuições também têm configuração de System V init modificadas para que se assemelhem ao estilo BSD.)

Há várias implementações diferentes de init porque o System V init e outras versões mais antigas eram baseadas em uma sequência que realizava somente uma tarefa de inicialização de cada vez. Nesse esquema, é relativamente fácil resolver dependências. No entanto o desempenho não é muito bom, pois duas partes da sequência de boot normalmente não podem executar ao mesmo tempo. Outra limitação está no fato de que você só pode iniciar um conjunto fixo de serviços, conforme definido pela sequência de boot: quando um novo hardware é conectado ou um serviço que ainda não esteja executando se torna necessário, não há uma maneira padronizada de coordenar os novos componentes com o init.

O systemd e o Upstart tentam remediar o problema do desempenho ao permitir que vários serviços iniciem em paralelo, agilizando assim o processo de boot. Suas implementações, porém, são bem diferentes:

- O systemd é orientado a objetivos. Você define uma meta que quer atingir, juntamente com suas dependências e quando quer alcançá-la. O systemd satisfaz às dependências e alcança a meta. Ele também pode atrasar o início de um serviço até que ele seja absolutamente necessário.
- O Upstart é reacionário. Ele recebe eventos e, de acordo com eles, executa jobs que, por sua vez, podem gerar mais eventos, fazendo o Upstart executar mais jobs e assim por diante.

Os sistemas de init systemd e Upstart também oferecem uma maneira mais sofisticada de iniciar e monitorar serviços. Em sistemas init tradicionais, espera-se que daemons de serviço iniciem a si mesmos a partir de scripts. Um script executa um programa daemon, que se desassocia do script e executa de maneira autônoma. Para encontrar o PID de um daemon de serviço, é preciso usar `ps` ou algum outro sistema específico para o serviço. Em contrapartida, o Upstart e o systemd podem administrar daemons individuais de serviço desde o início, dando mais capacidade ao usuário e permitindo-lhe ver de forma mais clara o que está exatamente sendo executado no sistema.

Como os novos sistemas `init` não são centrados em scripts, os serviços de configuração para eles também tendem a ser mais simples. Em particular, os scripts do System V `init` tendem a conter vários comandos semelhantes, concebidos para iniciar, finalizar e reiniciar serviços. Toda essa redundância não é necessária com o `systemd` e o `Upstart`, que permitem que você se concentre nos serviços propriamente ditos, em vez de focar em seus scripts.

Por fim, tanto o `systemd` quanto o `Upstart` oferecem algum nível de serviços por demanda. Em vez de tentar iniciar todos os serviços que possam ser necessários no momento do boot (como faria o System V `init`), eles iniciam alguns serviços somente quando esses forem necessários. Essa ideia não é realmente nova; isso era feito com o `daemon inetd` tradicional, porém as novas implementações são mais sofisticadas.

Tanto o `systemd` quanto o `Upstart` oferecem certo nível de compatibilidade para trás com o System V. Por exemplo, ambos suportam o conceito de `runlevels` (níveis de execução).

6.2 Os runlevels do System V

Em um instante qualquer em um sistema Linux, um determinado conjunto base de processos (como `crond` e `udev`) estará executando. No System V `init`, esse estado do computador é chamado de *runlevel* (nível de execução), que é representado por um número de 0 a 6. Um sistema permanece a maior parte de seu tempo em um único `runlevel`, porém, ao desligar o computador, o `init` muda para um `runlevel` diferente para finalizar os serviços do sistema de maneira organizada e para ordenar ao kernel que pare.

Você pode verificar o `runlevel` de seu sistema usando o comando `who -r`. Um sistema executando `Upstart` responderá com algo como:

```
$ who -r
run-level 2  2015-09-06 08:37
```

Essa saída nos informa que o `runlevel` atual é 2, além de mostrar a data e a hora em que o `runlevel` foi estabelecido.

Os `runlevels` servem a diversos propósitos, porém o mais comum é fazer a distinção entre os estados de inicialização do sistema, de desligamento e dos modos monousuário e console. Por exemplo, sistemas baseados em Fedora tradicionalmente usavam os `runlevels` de 2 a 4 para o console de texto; um `runlevel` igual a 5 significava que o sistema iria iniciar um login baseado em GUI.

Porém os `runlevels` estão se tornando um recurso do passado. Embora as três versões

de init descritas neste livro os suportem, o systemd e o Upstart consideram os runlevels obsoletos como estados finais do sistema. Para o systemd e o Upstart, os runlevels existem principalmente para iniciar serviços que suportem somente os scripts do System V init, e as implementações são tão diferentes que, mesmo estando familiarizado com um tipo de init, você não saberá necessariamente o que fazer com outro.

6.3 Identificando o seu init

Antes de prosseguir, você deve determinar a versão de init de seu sistema. Se não tiver certeza, verifique o seu sistema da seguinte maneira:

- Se o seu sistema tiver diretórios */usr/lib/systemd* e */etc/systemd*, você terá o systemd. Vá para a seção 6.4.
- Se você tiver um diretório */etc/init* que contenha vários arquivos *.conf*, provavelmente você estará executando o Upstart (a menos que você esteja executando o Debian 7, caso em que provavelmente você tem o System V init). Vá para a seção 6.5.
- Se nenhuma das opções anteriores for verdadeira, mas você tiver um arquivo */etc/inittab*, é provável que esteja executando o System V init. Vá para a seção 6.6.

Se o seu sistema tiver páginas de manual instaladas, consultar a página de manual `init(8)` deverá ajudá-lo a identificar a sua versão.

6.4 systemd

O systemd init é uma das implementações mais recentes de init do Linux. Além de cuidar do processo normal de boot, o systemd tem como objetivo incorporar vários serviços-padrão do Unix, como `cron` e `inetd`. Nesse sentido, ele se inspira um pouco no `launchd` da Apple. Um de seus recursos mais significativos é a capacidade de atrasar o início dos serviços e de recursos do sistema operacional até que eles sejam necessários.

Há tantos recursos no systemd que pode ser muito difícil saber em que ponto devemos começar para conhecer o básico. Vamos descrever o que acontece quando o systemd executa no momento do boot:

1. O systemd carrega a sua configuração.
2. O systemd determina o seu objetivo de boot, que normalmente se chama *default.target*.

3. O `systemd` determina todas as dependências do objetivo de boot default, as dependências dessas dependências e assim por diante.
4. O `systemd` ativa as dependências e o objetivo de boot.
5. Após o boot, o `systemd` pode reagir aos eventos do sistema (por exemplo, `uevents`) e ativar componentes adicionais.

Ao iniciar os serviços, o `systemd` não segue uma sequência rígida. Como ocorre com outros sistemas `init` modernos, há uma dose considerável de flexibilidade no processo de inicialização do `systemd`. A maioria das configurações de `systemd` tenta evitar deliberadamente qualquer tipo de sequência de inicialização, preferindo usar outros métodos para solucionar dependências estritas.

6.4.1 Unidades e tipos de unidade

Um dos aspectos mais interessantes a respeito do `systemd` é que ele não lida somente com processos e serviços; o `systemd` também pode montar sistemas de arquivos, monitorar sockets de rede, executar temporizadores e realizar outras tarefas. Cada tipo de capacidade é chamada de *tipo de unidade* (unit type), e cada capacidade específica é chamada de *unidade* (unit). Ao habilitar uma unidade, você estará *ativando-a*.

Em vez de descrever todos os tipos de unidade [eles podem ser encontrados na página de manual `systemd(1)`], você verá, a seguir, alguns dos tipos de unidade que executam as tarefas necessárias a qualquer sistema Unix no momento do boot.

- Unidades de serviço (service units) – controlam os tradicionais daemons de serviços em um sistema Unix.
- Unidades de montagem (mount units) – controlam a associação de sistemas de arquivos ao sistema.
- Unidades de alvo (target units) – controlam outras unidades, normalmente agrupando-as.

O objetivo de boot default geralmente é uma unidade de alvo que agrupa várias unidades de serviço e de montagem como dependências. Como resultado, é fácil obter um quadro parcial do que irá acontecer no boot, e você pode até mesmo criar um diagrama de árvore de dependências usando o comando `systemctl dot`. Você perceberá que a árvore é bem grande em um sistema típico porque muitas unidades não são executadas por padrão.

A figura 6.1 mostra uma parte da árvore de dependências para a unidade *default.target* de um sistema Fedora. Ao ativar essa unidade, todas as unidades abaixo dela na árvore

também serão ativadas.

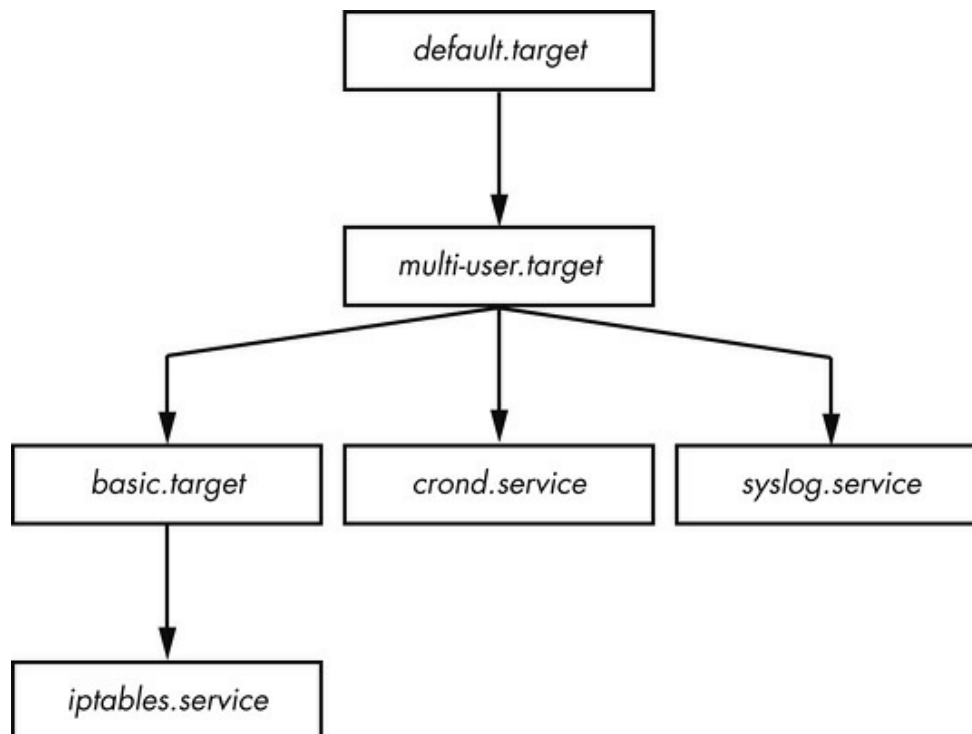


Figura 6.1 – Árvore de dependência de unidades.

6.4.2 Dependências do systemd

Dependências do momento de boot e dependências operacionais são mais complicadas do que parecem à primeira vista, pois dependências estritas são bem inflexíveis. Por exemplo, suponha um cenário em que você queira exibir um prompt de login após ter iniciado um servidor de banco de dados; você define uma dependência do prompt de login para o servidor de banco de dados. No entanto, se o servidor de banco de dados falhar, o prompt de login também falhará por causa dessa dependência, e você não será nem mesmo capaz de fazer login em seu computador para corrigir esse problema.

As tarefas do momento de boot do Unix são razoavelmente tolerantes a falhas e, com frequência, podem falhar sem causar problemas sérios aos serviços-padrão. Por exemplo, se um disco de dados de um sistema tiver sido removido, porém sua entrada em `/etc/fstab` tiver sido mantida, a montagem do sistema de arquivos inicial irá falhar. Entretanto essa falha normalmente não afetará seriamente a operação-padrão do sistema operacional.

Para acomodar a necessidade de ter flexibilidade e tolerância a falhas, o systemd oferece uma variedade de tipos e de estilos de dependência. Iremos nomeá-las de acordo com a sintaxe de sua palavra-chave, porém não entraremos em detalhes sobre a sintaxe da configuração até a seção 6.4.3. Inicialmente, vamos dar uma olhada nos tipos

básicos:

- **Requires** – dependências estritas. Ao ativar uma unidade com uma dependência **Requires**, o `systemd` tenta ativar a unidade de dependência. Se a unidade de dependência falhar, o `systemd` desativará a unidade dependente.
- **Wants** – dependências somente para ativação. Ao ativar uma unidade, o `systemd` ativará as dependências **Wants** da unidade, porém não se importará se essas dependências falharem.
- **Requisite** – unidades que já deverão estar ativas. Antes de ativar uma unidade com uma dependência **Requisite**, o `systemd` irá verificar antes o status da dependência. Se a dependência não tiver sido ativada, o `systemd` falhará na ativação da unidade com a dependência.
- **Conflicts** – dependências negativas. Ao ativar uma unidade com uma dependência **Conflict**, o `systemd` desativará automaticamente a dependência se ela estiver ativa. A ativação simultânea de duas unidades conflitantes causará uma falha.

👍 **Observação:** o tipo de dependência **Wants** é especialmente significativo porque ele não propaga falhas para outras unidades. A documentação do `systemd` afirma que essa é a maneira pela qual você deve especificar dependências, se possível, e é fácil ver por quê. Esse comportamento produz um sistema muito mais robusto, semelhante àquele de um `init` tradicional.

Você sempre pode associar dependências “invertidas”. Por exemplo, para adicionar a Unidade A como uma dependência **Wants** da Unidade B, não é preciso adicionar **Wants** na configuração da Unidade B. Em vez disso, você pode instalá-la como um **WantedBy** na configuração da Unidade A. O mesmo é válido para a dependência **RequiredBy**. A configuração para (e o resultado de) uma dependência “By” é um pouco mais intrincada do que simplesmente editar um arquivo de configuração; veja a seção “Habilitando unidades e a seção [Install]”.

Você pode ver as dependências de uma unidade usando o comando `systemctl`, desde que um tipo de dependência seja especificado, por exemplo, **Wants** ou **Requires**:

```
# systemctl show -p tipo unidade
```

Ordenação

Nenhuma das sintaxes de dependência vistas até agora especifica explicitamente a ordem. Por padrão, ativar uma unidade com **Requires** ou **Wants** faz com que o `systemd` ative todas essas dependências ao mesmo tempo como a primeira unidade. Isso é ótimo, pois você quer iniciar o máximo de serviços o mais rápido possível para reduzir o tempo de boot. Entretanto há situações em que uma unidade deverá iniciar depois de outra. Por exemplo, no sistema em que a figura 6.1 está baseada, a unidade

default.target está definida para iniciar após *multi-user.service* (essa distinção na ordem não está sendo mostrada na figura).

Para ativar as unidades em uma ordem em particular, os seguintes modificadores de dependência podem ser usados:

- **Before** – a unidade corrente será ativada antes da(s) unidade(s) listada(s). Por exemplo, se *Before=bar.target* aparecer em *foo.target*, o systemd ativará *foo.target* antes de *bar.target*.
- **After** – a unidade corrente será ativada após a(s) unidade(s) listada(s).

Dependências condicionais

Várias palavras-chave de condição de dependência funcionam em vários estados de operação do sistema no lugar de unidades do systemd. Por exemplo:

- **ConditionPathExists=*p***: – verdadeiro se o path *p* (de arquivo) existir no sistema.
- **ConditionPathIsDirectory=*p***: – verdadeiro se *p* for um diretório.
- **ConditionFileNotEmpty=*p***: – verdadeiro se *p* for um arquivo e seu tamanho for diferente de zero.

Se uma dependência condicional em uma unidade for falsa quando systemd tentar ativar a unidade, ela não será ativada, embora isso se aplique somente à unidade em que a dependência aparecer. Desse modo, se você ativar uma unidade que tenha uma dependência de condição bem como outras dependências de unidade, o systemd tentará ativar as dependências de unidade, independentemente de a condição ser verdadeira ou falsa.

Outras dependências são principalmente variações das anteriores. Por exemplo, a dependência *RequiresOverridable* é exatamente como *Requires* quando executada normalmente, porém atua como uma dependência *Wants* se uma unidade for ativada manualmente. [Para ver uma lista completa, consulte a página de manual `systemd.unit(5)`.]

Agora que você já viu algumas partes da configuração do systemd, vamos dar uma olhada em alguns arquivos de unidade e ver como trabalhar com eles.

6.4.3 Configuração do systemd

Os arquivos de configuração do systemd estão espalhados em vários diretórios pelo sistema, portanto, normalmente, você não encontrará os arquivos para todas as unidades de um sistema em um só local. Apesar do que foi dito, há dois diretórios principais

para a configuração do systemd: o diretório de *unidades do sistema* (globalmente configurado – normalmente é `/usr/lib/systemd/system`) e um diretório de *configuração do sistema* (definições locais – normalmente é `/etc/systemd/system`).

Para evitar confusão, atenha-se a esta regra: evite fazer alterações no diretório de unidades do sistema porque a sua distribuição irá mantê-lo para você. Faça suas alterações locais no diretório de configuração do sistema. Portanto, quando tiver a opção entre modificar algo em `/usr` e em `/etc`, sempre altere em `/etc`.

👉 **Observação:** você pode verificar o path atual de pesquisa da configuração do systemd (incluindo as precedências) usando o comando a seguir:

```
# systemctl -p UnitPath show
```

Entretanto essa configuração em particular é proveniente de uma terceira fonte: as configurações de `pkg-config`. Para ver os diretórios de unidade e de configuração de seu sistema, utilize os comandos a seguir:

```
$ pkg-config systemd --variable=systemdsystemunitdir
```

```
$ pkg-config systemd --variable=systemdsystemconfdir
```

Arquivos de unidade

Os arquivos de unidade são derivados da XDG Desktop Entry Specification (para arquivos `.desktop`, que são bem semelhantes aos arquivos `.ini` de sistemas Microsoft), com nomes de seção entre colchetes (`[]`), além de variáveis e atribuições de valores (opções) em cada seção.

Considere o exemplo de arquivo de unidade `media.mount` em `/usr/lib/systemd/system`, que é padrão em instalações Fedora. Esse arquivo representa o sistema de arquivos `tmpfs` `/media`, que é um diretório contêiner para montagem de mídia removível.

```
[Unit]
Description=Media Directory
Before=local-fs.target

[Mount]
What=tmpfs
Where=/media
Type=tmpfs
Options=mode=755,nosuid,nodev,noexec
```

Há duas seções nesse arquivo. A seção `[Unit]` fornece alguns detalhes sobre a unidade e contém informações de descrição e de dependência. Em particular, essa unidade está definida para ser ativada antes da unidade `local-fs.target`.

A seção `[Mount]` detalha a unidade como sendo uma unidade de montagem, e dá detalhes sobre o ponto de montagem, o tipo do sistema de arquivos e as opções de montagem, conforme descritas na seção 4.2.6. A variável `What=` identifica o dispositivo ou o UUID

do dispositivo a ser montado. Nesse caso, está definido como `tmpfs`, pois esse sistema de arquivos não tem um dispositivo. [Para ver uma lista completa das opções de unidade de montagem, consulte a página de manual `systemd.mount(5)`.]

Vários outros arquivos de configuração de unidades são igualmente simples. Por exemplo, o arquivo de unidade de serviços `sshd.service` permite logins em shell seguro (secure shell):

```
[Unit]
Description=OpenSSH server daemon
After=syslog.target network.target auditd.service

[Service]
EnvironmentFile=/etc/sysconfig/ssh
ExecStartPre=/usr/sbin/ssh-keygen
ExecStart=/usr/sbin/ssh -D $OPTIONS
ExecReload=/bin/kill -HUP $MAINPID

[Install]
WantedBy=multi-user.target
```

Como esse é um alvo de serviço, você encontrará os detalhes do serviço na seção `[Service]`, incluindo o modo de preparar, iniciar e recarregar o serviço. Você encontrará uma listagem completa na página de manual `systemd.service(5)` (e em `systemd.exec(5)`), bem como na discussão sobre monitoração de processos na seção 6.4.6.

Habilitando unidades e a seção `[Install]`

A seção `[Install]` no arquivo de unidade `sshd.service` é importante porque ela nos ajuda a entender como usar as opções de dependência `WantedBy` e `RequiredBy` do `systemd`. Na realidade, esse é um sistema para habilitar unidades sem modificar nenhum arquivo de configuração. Durante a operação normal, o `systemd` ignora a seção `[Install]`. Entretanto considere o caso em que `sshd.service` está desabilitado em seu sistema e você gostaria de habilitá-lo. Ao *habilitar* uma unidade, o `systemd` lê a seção `[Install]`; nesse caso, habilitar a unidade `sshd.service` faz com que o `systemd` veja a dependência `WantedBy` para *multi-user.target*. Em resposta, o `systemd` cria um link simbólico para `sshd.service` no diretório de configuração do sistema da seguinte maneira:

```
ln -s '/usr/lib/systemd/system/ssh.service' '/etc/systemd/system/multi-user.target.wants/ssh.service'
```

Observe que o link simbólico é colocado em um subdiretório que corresponde à unidade dependente (*multi-user.target*, nesse caso).

A seção `[Install]` normalmente é responsável pelos diretórios *.wants* e *.requires* no diretório de configuração do sistema (`/etc/systemd/system`). No entanto há também diretórios *.wants* no diretório de configuração de unidades (`/usr/lib/systemd/system`), e

you can also add links that do not correspond to sections [Install] in the unit files. These manual additions are a simple way to add a dependency without modifying a unit file that may be overwritten in the future (for example, by a software upgrade).

 **Observação:** enabling a unit is different from activating it. When you enable a unit, you are installing it in the systemd configuration, making semi-permanent changes that will survive a system reboot. However, you do not always need to enable a unit explicitly. If the unit file has an [Install] section, you should enable it with `systemctl enable`; otherwise, the existence of the file is sufficient to enable it. When you start a unit with `systemctl start`, you are simply enabling it in the current execution environment. In addition, enabling a unit does not activate it.

Variáveis e especificadores

The unit file `sshd.service` also shows the use of variables – specifically, the environment variables `$OPTIONS` and `$MAINPID` that are passed to `systemd`. `$OPTIONS` are options that you can pass to `sshd` when the unit is activated with `systemctl`, and `$MAINPID` is the process monitored by the service (see section 6.4.6).

A *specifier* is another resource similar to variables, found with frequency in unit files. Specifiers start with a percent sign (%). For example, the specifier `%n` is the name of the current unit, and the specifier `%H` is the name of the current host.

 **Observação:** the name of the unit can contain some interesting specifiers. You can parametrize a single unit file to generate several copies of a service, for example, processes `getty` running on `tty1`, `tty2` and so on. To use these specifiers, add the symbol `@` at the end of the unit name. For `getty`, create a unit file called `getty@.service`, which allows you to refer to the units as `getty@tty1` and `getty@tty2`. Everything after `@` is called an *instance*, and when the unit file is processed, `systemd` will expand the specifier `%I` with an instance. You will see this in action with the unit files `getty@.service`, which are used by most distributions that use `systemd`.

6.4.4 Operação do systemd

You will interact with `systemd` primarily through the `systemctl` command, which allows you to activate and deactivate services, list status, reload the configuration and much more.

The most important basic commands relate to the retrieval of information about units. For example, to see a list of active units in your system, run the `list-units` command. (This, in fact, is the default command for `systemctl`, so the `list-units` part is not really necessary.)

```
$ systemctl list-units
```

O formato da saída é típico de um comando Unix de listagem de informações. Por exemplo, o cabeçalho e a linha para *media.mount* têm um aspecto semelhante a:

```
UNIT          LOAD   ACTIVE SUB    JOB DESCRIPTION
media.mount    loaded active mounted Media Directory
```

Esse comando gera muitos dados de saída, pois um sistema típico tem inúmeras unidades ativas, porém essa saída ainda será resumida, pois `systemctl` trunca qualquer nome de unidade que seja realmente longo. Para ver os nomes completos das unidades, utilize a opção `--full`, e para ver todas as unidades (e não apenas as ativas), use a opção `--all`.

Uma operação particularmente útil de `systemctl` consiste em obter o status de uma unidade. Por exemplo, aqui está um comando típico de status e a sua saída:

```
$ systemctl status media.mount
```

```
media.mount - Media Directory
Loaded: loaded (/usr/lib/systemd/system/media.mount; static)
Active: active (mounted) since Wed, 13 May 2015 11:14:55 -0800;
37min ago
Where: /media
What: tmpfs
Process: 331 ExecMount=/bin/mount tmpfs /media -t tmpfs -o mode=755,nosuid,nodev,noexec
(code=exited, status=0/SUCCESS)
CGroup: name=systemd/system/media.mount
```

Observe que há muito mais informações de saída nesse caso em comparação ao que você veria em qualquer sistema `init` tradicional. Você tem não só o estado da unidade, mas também o comando exato utilizado para efetuar a montagem, seu PID e o status de saída.

Uma das partes mais interessantes da saída é o nome do grupo de controle. No exemplo anterior, o grupo de controle não inclui nenhuma informação além do nome `systemd/system/media.mount` porque os processos da unidade já terminaram. No entanto, se você obtiver o status de uma unidade de serviço como *NetworkManager.service*, você também verá a árvore de processos do grupo de controle. Os grupos de controle sem o restante do status da unidade podem ser visualizados por meio do comando `systemd-cgls`. Você aprenderá mais sobre grupos de controle na seção 6.4.6.

O comando de status também exibe informações recentes do *journal* da unidade (um log que registra informações de diagnóstico para cada unidade). O *journal* completo de uma unidade pode ser visto com o comando a seguir:

```
$ journalctl _SYSTEMD_UNIT=unidade
```

(Essa sintaxe causa um pouco de estranheza porque `journalctl` pode acessar outros logs além daqueles de uma unidade do `systemd`.)

Para ativar, desativar e reiniciar unidades, utilize os comandos `systemd start`, `stop` e `restart`. Porém, se um arquivo de configuração de unidade for alterado, você poderá dizer ao `systemd` para recarregar o arquivo usando uma de duas opções:

- `systemctl reload unidade` – recarrega somente a configuração para *unidade*.
- `systemctl daemon-reload` – recarrega a configuração de todas as unidades.

Solicitações para ativar, reativar e reiniciar unidades são conhecidas como *jobs* no `systemd`, e elas são essencialmente mudanças de estado de unidades. Você pode verificar os jobs correntes em um sistema usando:

```
$ systemctl list-jobs
```

Se um sistema estiver executando durante algum tempo, você poderá esperar razoavelmente que não haja nenhum job ativo, pois todas as ativações deverão ter sido concluídas. Entretanto, no momento do boot, às vezes, você poderá efetuar login rápido o suficiente para ver algumas unidades iniciarem de modo tão lento que elas ainda não estarão totalmente ativas. Por exemplo:

JOB UNIT	TYPE	STATE
1 graphical.target	start	waiting
2 multi-user.target	start	waiting
71 systemd-...nlevel.service	start	waiting
75 sm-client.service	start	waiting
76 sendmail.service	start	running
120 systemd-...ead-done.timer	start	waiting

Nesse caso, o job 76 – a inicialização da unidade *sendmail.service* – está realmente demorando bastante. Os outros jobs listados estão em estado de espera, muito provavelmente porque estão esperando pelo job 76. Quando *sendmail.service* acabar de iniciar e estiver totalmente ativo, o job 76 estará completo, o restante dos jobs também estarão completos e a lista de jobs ficará vazia.

 **Observação:** o termo *job* pode ser confuso, especialmente porque o Upstart – outro sistema init descrito neste capítulo – utiliza a palavra *job* para se referir (grosseiramente) àquilo que o `systemd` chama de unidade. É importante lembrar-se de que embora um job do `systemd` associado a uma unidade vá terminar, a unidade propriamente dita poderá estar ativa e em execução depois, especialmente no caso das unidades de serviço.

Consulte a seção 6.7 para saber como desligar e reiniciar o sistema.

6.4.5 Adicionando unidades ao `systemd`

Adicionar unidades ao `systemd` é principalmente uma questão de criar e, em seguida,

ativar e, possivelmente, habilitar arquivos de unidade. Normalmente, você deverá colocar seus próprios arquivos de unidade no diretório de configuração do sistema – */etc/systemd/system* – para que você não os confunda com nada que tenha vindo com a sua distribuição, e para que a distribuição não os sobrescreva quando você fizer um upgrade.

Como é fácil criar unidades de alvo que não façam nada e que não interfiram em nada, você deve tentar fazer isso. Eis o modo de criar dois alvos, um com uma dependência em relação ao outro:

1. Crie um arquivo de unidade chamado *test1.target*:

```
[Unit]
Description=test 1
```

2. Crie um arquivo *test2.target* com uma dependência de *test1.target*:

```
[Unit]
Description=test 2
Wants=test1.target
```

3. Ative a unidade *test2.target* (lembre-se de que a dependência em *test2.target* faz com que o systemd ative *test1.target* quando você fizer isso):

```
# systemctl start test2.target
```

4. Confira se ambas as unidades estão ativas:

```
# systemctl status test1.target test2.target
test1.target - test 1
    Loaded: loaded (/etc/systemd/system/test1.target; static)
    Active: active since Thu, 12 Nov 2015 15:42:34 -0800; 10s ago

test2.target - test 2
    Loaded: loaded (/etc/systemd/system/test2.target; static)
    Active: active since Thu, 12 Nov 2015 15:42:34 -0800; 10s ago
```

 **Observação:** se o seu arquivo de unidade tiver uma seção [Install], “habilite” a unidade antes de ativá-la:

```
# systemctl enable unidade
```

Tente fazer isso com o exemplo anterior. Remova a dependência de *test2.target* e adicione uma seção [Install] em *test1.target* contendo *WantedBy=test2.target*.

Removendo unidades

Para remover uma unidade, siga estes passos:

1. Desative a unidade, se for necessário:

```
# systemctl stop unidade
```

2. Se a unidade tiver uma seção [Install], desabilite a unidade para remover qualquer link simbólico dependente:

```
# systemctl disable unidade
```

3. Remova o arquivo de unidade, se quiser.

6.4.6 Monitoração de processos e sincronização no systemd

O systemd deseja ter uma quantidade razoável de informações e de controle sobre todos os processos que ele iniciar. O principal problema com que ele se depara é o fato de que um serviço pode iniciar de diferentes maneiras: ele pode fazer um fork e gerar novas instâncias de si mesmo ou até mesmo se transformar em um daemon e se desconectar do processo original.

Para minimizar o trabalho que um desenvolvedor de pacotes ou um administrador deve fazer para criar um arquivo de unidade funcional, o systemd utiliza *grupos de controle* (cgroups) – um recurso opcional do kernel do Linux que permite uma monitoração mais detalhada da hierarquia de um processo. Se você já trabalhou anteriormente com o Upstart, saberá que você precisará de um pouco de trabalho extra para descobrir qual é o processo principal de um serviço. No systemd, não é preciso se preocupar em saber quantas vezes um processo realizou fork – somente é necessário saber se houve fork. Utilize a opção `Type` em seu arquivo de unidade de serviço para indicar o seu comportamento de inicialização. Há dois estilos básicos de inicialização:

- `Type=simple` – o processo do serviço não realiza fork.
- `Type=forking` – o serviço realiza fork e o systemd espera que o processo original do serviço termine. No término, o systemd supõe que o serviço esteja pronto.

A opção `Type=simple` não leva em consideração o fato de que um serviço pode demorar um pouco para ser instalado, e o systemd não sabe quando deve iniciar qualquer dependência que exija terminantemente que um serviço como esse esteja pronto. Uma maneira de lidar com isso é usar inicializações tardias (veja a seção 6.4.7). Entretanto alguns estilos de inicialização `Type` podem indicar que o próprio serviço irá notificar o systemd quando estiver pronto:

- `Type=notify` – o serviço envia uma notificação específica ao systemd [por meio da chamada de função `sd_notify()`] quando estiver pronto.
- `Type=dbus` – o serviço se registra no D-bus (Desktop Bus) quando estiver pronto.

Outro estilo de inicialização de serviço é especificado com `Type=oneshot`; nesse caso, o processo do serviço, na verdade, termina completamente quando tiver finalizado. Em

um serviço desse tipo, é quase certo que você deverá acrescentar uma opção `RemainAfterExit=yes` para que o `systemd` continue a considerar o serviço como ativo, mesmo após seus processos terem terminado.

Por fim, há um último estilo: `Type=idle`. Essa opção simplesmente instrui o `systemd` a não iniciar o serviço até que não haja nenhum job ativo. A ideia, nesse caso, é simplesmente atrasar o início de um serviço até que os outros serviços tenham iniciado para manter a carga do sistema baixa ou evitar que os serviços interfiram na saída uns dos outros. (Lembre-se de que, depois que um serviço tiver iniciado, o job do `systemd` que iniciou o serviço termina.)

6.4.7 `systemd` por demanda e inicialização com recursos paralelos

Um dos recursos mais significativos do `systemd` é sua capacidade de atrasar a inicialização de uma unidade até que ela seja absolutamente necessária. A configuração normalmente funciona do seguinte modo:

1. Crie uma unidade do `systemd` (vamos chamá-la de Unidade A) para o serviço de sistema que você queira disponibilizar, conforme feito normalmente.
2. Identifique um recurso do sistema, como uma porta de rede/socket, um arquivo ou um dispositivo que a Unidade A utilize para oferecer seus serviços.
3. Crie outra unidade do `systemd` – a Unidade R – para representar esse recurso. Essas unidades têm tipos especiais, como unidades de socket, unidades de path e unidades de dispositivo.

Operacionalmente, o que ocorre é:

1. Na ativação da Unidade R, o `systemd` monitora o recurso.
2. Quando houver uma tentativa de acessar o recurso, o `systemd` o bloqueará e a entrada para o recurso é bufferizada.
3. O `systemd` ativa a Unidade A.
4. Quando o serviço da Unidade A estiver pronto, ele assumirá o controle do recurso, lerá a entrada bufferizada e executará normalmente.

Há algumas preocupações:

- Você deve garantir que a unidade de seu recurso incluirá todos os recursos que o serviço disponibilizar. Isso normalmente não é um problema, pois a maioria dos serviços tem somente um ponto de acesso.
- Você deve garantir que a unidade de seu recurso esteja associada à unidade de

serviço que ela representa. Isso pode ser implícito ou explícito e, em alguns casos, muitas opções representam diferentes maneiras de o `systemd` realizar a passagem para a unidade de serviço.

- Nem todos os servidores sabem como fazer a interface com as unidades que o `systemd` pode disponibilizar.

Se você já sabe o que utilitários como `inetd`, `xinetd` e `automount` fazem, verá que há muitas semelhanças. Realmente, o conceito não é nada novo [e, de fato, o `systemd` inclui suporte para unidades de montagem automática (`automount`)]. Veremos na seção “Um exemplo de unidade e de serviço de socket”. Porém vamos inicialmente dar uma olhada em como essas unidades de recurso ajudam você no momento do boot.

Otimização de boot com unidades auxiliares

Um estilo comum de ativação de unidade no `systemd` tenta simplificar a ordem de dependências e agilizar o momento do boot. É semelhante a uma inicialização por demanda, no sentido de que uma unidade de serviço e uma unidade auxiliar representam o recurso oferecido pela unidade de serviço, exceto pelo fato de que, nesse caso, o `systemd` inicia a unidade de serviço assim que ele ativa a unidade auxiliar.

O raciocínio por trás desse esquema é que as unidades de serviço essenciais no momento do boot, como *syslog* e *dbus*, demoram um pouco para iniciar, e várias outras unidades dependem delas. Entretanto o `systemd` pode oferecer um recurso essencial de uma unidade (como uma unidade de socket) muito rapidamente e, em seguida, poderá ativar imediatamente não apenas a unidade essencial como também qualquer unidade que dependa do recurso essencial. Depois que a unidade essencial estiver pronta, ela assumirá o controle do recurso.

A figura 6.2 mostra como isso pode funcionar em um sistema tradicional. Na linha de tempo desse boot, o Serviço E disponibiliza um Recurso R essencial. Os Serviços A, B e C dependem desse recurso e devem esperar até que o Serviço E tenha iniciado. Durante a inicialização, o sistema demora bastante para começar a iniciar o Serviço C.

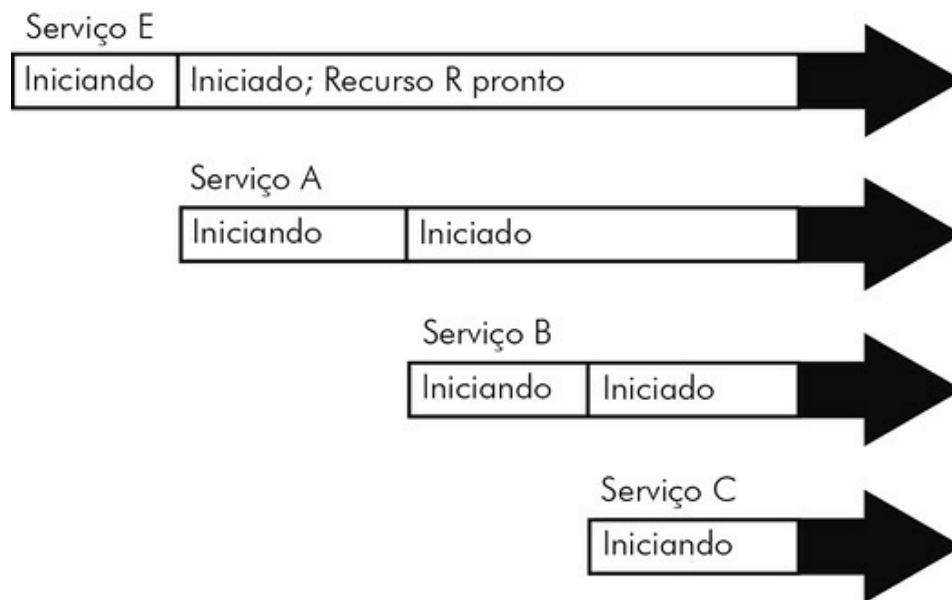


Figura 6.2 – Linha do tempo sequencial de inicialização, com dependência de um recurso.

A figura 6.3 mostra uma configuração de boot equivalente do systemd. Os serviços estão representados pelas Unidades A, B, C e E, e uma nova Unidade R representa o recurso disponibilizado pela Unidade E. Como o systemd pode disponibilizar uma interface para a Unidade R enquanto a Unidade E inicia, as Unidades A, B, C e E podem ser todas iniciadas ao mesmo tempo. A Unidade E assume o controle pela Unidade R quando ela estiver pronta. (Um ponto interessante, nesse caso, é que as Unidades A, B e C podem não precisar acessar explicitamente a Unidade R antes de terminarem suas inicializações, como demonstrado pela Unidade B na figura.)

👍 **Observação:** quando a inicialização é paralelizada dessa maneira, há uma chance de o seu sistema ficar temporariamente mais lento por causa da grande quantidade de unidades iniciando ao mesmo tempo.

O ponto a ser lembrado é que, embora você não esteja criando uma inicialização de unidades por demanda nesse caso, você estará usando os mesmos recursos que possibilitam esse tipo de inicialização. Para ver exemplos comuns da vida real, consulte as unidades de configuração de syslog e de D-Bus em um computador que esteja executando systemd; é muito provável que elas estejam paralelizadas dessa maneira.

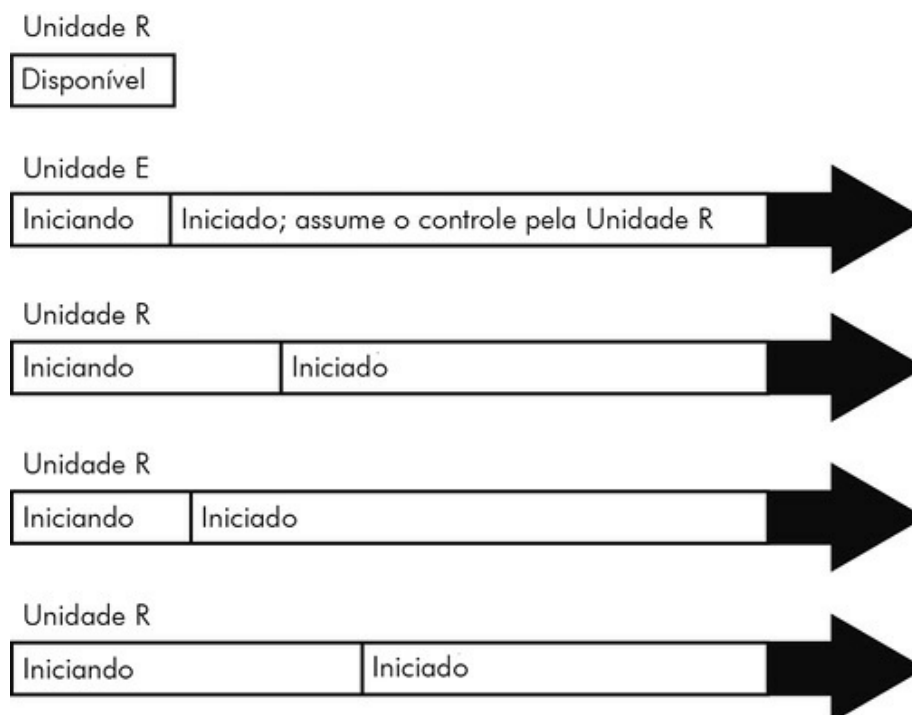


Figura 6.3 – Linha do tempo de boot do systemd, com uma unidade de recurso.

Um exemplo de unidade e de serviço de socket

Vamos agora dar uma olhada em um exemplo: um serviço simples de eco de rede que usa uma unidade de socket. É um material, de certo modo, avançado, e você poderá não entendê-lo realmente até ter lido a discussão sobre TCP, portas e ouvir portas no capítulo 9, e sobre sockets no capítulo 10, portanto sinta-se à vontade para pular essa parte e retornar depois.

A ideia por trás desse serviço é que quando um cliente de rede se conectar ao serviço, esse repetirá tudo o que o cliente enviar. A unidade ficará ouvindo a porta TCP 22222. Nós a chamaremos de echo service (serviço de eco) e começaremos com uma unidade de socket, representada pelo arquivo de unidade *echo.socket* a seguir:

```
[Unit]
Description=echo socket

[Socket]
ListenStream=22222
Accept=yes
```

Observe que não há nenhuma menção à unidade de serviço que esse socket suporta no arquivo de unidade. Então qual é o arquivo de unidade de serviço correspondente?

Seu nome é *echo@.service*. A ligação é feita por convenção de nomenclatura; se um arquivo de unidade de serviço tiver o mesmo prefixo que um arquivo *.socket* (nesse caso, *echo*), o systemd saberá que deve ativar essa unidade de serviço quando houver

atividade na unidade de socket. Nesse caso, o `systemd` criará uma instância de *echo@.service* quando houver atividade em *echo.socket*.

Aqui está o arquivo de unidade *echo@.service*:

```
[Unit]
Description=echo service

[Service]
ExecStart=-/bin/cat
StandardInput=socket
```

👍 **Observação:** se você não gostar da ativação implícita de unidades baseada nos prefixos, ou houver necessidade de criar um sistema de ativação entre duas unidades com prefixos diferentes, uma opção explícita poderá ser usada na unidade que define o seu recurso. Por exemplo, use `Socket=bar.socket` em *foo.service* para fazer com que *bar.socket* passe o seu socket para *foo.service*.

Para fazer com que esse exemplo de unidade de serviço execute, você deve iniciar a unidade *echo.socket* por trás dela, desta maneira:

```
# systemctl start echo.socket
```

Agora você poderá testar o serviço ao se conectar com a sua porta local 22222. Quando o comando `telnet` a seguir fizer a conexão, digite qualquer dado e tecla Enter. O serviço repetirá de volta o que você digitar:

```
$ telnet localhost 22222
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hi there.
Hi there.
```

Quando você se sentir entediado com isso, tecla `Ctrl-]` em uma linha e, em seguida, `Ctrl-D`. Para interromper o serviço, finalize a unidade de socket:

```
# systemctl stop echo.socket
```

Instâncias e handoff

Pelo fato de a unidade *echo@.service* suportar várias instâncias simultâneas, há um `@` no nome (lembre-se de que, de acordo com a página 159, `@` significa parametrização). Por que você iria querer ter várias instâncias? O motivo para isso é que pode haver mais de um cliente de rede conectado ao serviço ao mesmo tempo e cada conexão deverá ter a sua própria instância.

Nesse caso, a unidade de serviço *deve* suportar várias instâncias por causa da opção `Accept` em *echo.socket*. Essa opção instrui o `systemd` não só a ouvir a porta, mas também a aceitar conexões de entrada e passar essas conexões para a unidade de serviço, com

cada conexão sendo uma instância separada. Cada instância lê dados da conexão como entrada-padrão, porém ela não necessariamente precisa saber que os dados estão vindo de uma conexão de rede.

👍 **Observação:** a maioria das conexões de rede exige mais flexibilidade do que somente um simples gateway para a entrada e a saída padrão, portanto não espere ser capaz de criar serviços de rede com um arquivo de unidade de serviço como o arquivo *echo@.service* mostrado aqui.

Embora a unidade de serviço pudesse fazer todo o trabalho de aceitar a conexão, ela não teria o *@* em seu nome se o fizesse. Nesse caso, ela assumiria o controle completo do socket, e o *systemd* não tentaria ouvir a porta da rede novamente até que a unidade de serviço tivesse finalizado.

Os vários recursos e opções diferentes para efetuar o handoff (passagem) para as unidades de serviço dificultam fornecer um resumo categórico. Além do mais, a documentação sobre as opções está espalhada em várias páginas de manual. As páginas a serem verificadas para as unidades orientadas a recursos são: *systemd.socket(5)*, *systemd.path(5)* e *systemd.device(5)*. Um documento que frequentemente é menosprezado para unidades de serviço é *systemd.exec(5)*, que contém informações sobre como a unidade de serviço pode esperar receber um recurso na ativação.

6.4.8 Compatibilidade do *systemd* com o System V

Um recurso que distingue o *systemd* de outros sistemas *init* da mais nova geração é que ele tenta realizar um trabalho mais completo de monitoração de serviços iniciados pelos scripts de *init* compatíveis com System V. Isso funciona da seguinte maneira:

1. Inicialmente, o *systemd* ativa *runlevel<N>.target*, em que *N* é o runlevel.
2. Para cada link simbólico em */etc/rc<N>.d*, o *systemd* identifica o script em */etc/init.d*.
3. O *systemd* associa o nome do script a uma unidade de serviço (por exemplo, */etc/init.d/foo* seria *foo.service*).
4. O *systemd* ativa a unidade de serviço e executa o script com um argumento *start* ou *stop*, de acordo com o seu nome em *rc<N>.d*.
5. O *systemd* tenta associar qualquer processo do script com a unidade de serviço.

Como o *systemd* faz a associação com um nome de unidade de serviço, o *systemctl* pode ser usado para reiniciar o serviço ou para visualizar o seu status. Mas não espere nenhum milagre do modo de compatibilidade com System V; os scripts de *init* continuam tendo que ser executados serialmente, por exemplo.

6.4.9 Programas auxiliares do systemd

Ao começar a trabalhar com o systemd, você poderá perceber o número excepcionalmente grande de programas em */lib/systemd*. São principalmente programas de suporte para as unidades. Por exemplo, o *udev* faz parte do systemd, e você o encontrará aí como *systemd-udev*. Outro programa, o *systemd-fsck*, funciona como um intermediário entre o systemd e o *fsck*.

Muitos desses programas existem porque contêm sistemas de notificação que estão ausentes nos utilitários-padrão do sistema. Com frequência, eles simplesmente executam os utilitários-padrão do sistema e notificam o systemd sobre os resultados. (Afinal de contas, seria tolo tentar reimplementar todo o *fsck* no interior do systemd.)

👍 **Observação:** outro aspecto interessante desses programas é que eles estão escritos em C, pois um dos objetivos do systemd é reduzir a quantidade de shell scripts em um sistema. Há certo volume de debates para saber se fazer isso é uma boa ideia (afinal de contas, muitos desses programas provavelmente poderiam ter sido escritos como shell scripts), porém, desde que tudo funcione e faça isso de forma confiável, segura e razoavelmente rápida, há poucos motivos para se dar ao trabalho de tomar partido.

Quando você vir um programa em */lib/systemd* que não puder identificar, consulte a página de manual. Há uma boa chance de que essa página descreverá não só o utilitário, mas também o tipo de unidade que deverá ser expandido.

Se você não estiver executando (ou não estiver interessado no) Upstart, pule para a seção 6.6 para obter uma visão geral do processo System V init.

6.5 Upstart

A versão Upstart do init gira em torno de *jobs* e de *eventos*. Os jobs são ações realizadas na inicialização e em tempo de execução que o Upstart executa (por exemplo, serviços de sistema e configuração), enquanto eventos são mensagens que o Upstart recebe de si mesmo ou de outros processos (como *udev*). O Upstart funciona iniciando jobs em resposta a eventos.

Para ter uma ideia de como isso funciona, considere o job *udev* para iniciar o daemon *udev*. Seu arquivo de configuração normalmente é */etc/init/udev.conf*, que inclui o seguinte:

```
start on virtual-filesystems
stop on runlevel [06]
```

Essas linhas querem dizer que o Upstart inicia o job *udev* ao receber o evento *virtual-filesystems*, e finaliza o job ao receber um evento de runlevel com um argumento igual a 0 ou 6.

Há muitas variações nos eventos e em seus argumentos. Por exemplo, o Upstart pode reagir a eventos gerados em resposta a status de jobs, como o evento `started udev`, gerado pelo job *udev* mencionado anteriormente. Porém, antes de explicar os jobs em detalhes, apresentaremos a seguir uma visão geral de como o Upstart funciona.

6.5.1 Procedimento de inicialização do Upstart

Na inicialização, o Upstart faz o seguinte:

1. Carrega a sua configuração e os arquivos de configuração de jobs em */etc/init*.
2. Gera o evento `startup`.
3. Executa os jobs configurados para iniciar ao receber o evento `startup`.
4. Esses jobs iniciais geram seus próprios eventos, disparando mais jobs e eventos.

Ao finalizar todos os jobs associados a uma inicialização normal, o Upstart continua a monitorar e a reagir a eventos durante todo o tempo de execução do sistema.

A maioria das instalações com Upstart executa da seguinte maneira:

1. O job mais significativo que o Upstart executa em resposta ao evento `startup` é *mountall*. Esse job faz a associação dos sistemas de arquivos locais e virtuais ao sistema em execução no momento para que tudo o mais possa executar.
2. O job *mountall* gera vários eventos, incluindo `filesystem`, `virtual-filesystems`, `local-filesystems`, `remote-filesystems` e `all-swaps`, entre outros. Esses eventos indicam que os sistemas de arquivos importantes do sistema agora estão associados e prontos.
3. Em resposta a esses eventos, o Upstart inicia vários jobs de serviço essenciais. Por exemplo, o *udev* é iniciado em resposta ao evento `virtual-filesystems`, e *dbus*, em resposta ao evento `local-filesystems`.
4. Entre os jobs de serviço essenciais, o Upstart inicia o job *network-interfaces*, normalmente em resposta ao evento `local-filesystems` e ao fato de *udev* estar pronto.
5. O job *network-interfaces* gera o evento `static-network-up`.
6. O Upstart executa o job *rc-sysinit* em resposta aos eventos `filesystem` e `static-network-up`. Esse job é responsável por manter o runlevel corrente do sistema, e quando iniciado pela primeira vez sem um runlevel, ele alterna o sistema para o runlevel default ao gerar um evento `runlevel`.
7. O Upstart executa a maior parte dos demais jobs de inicialização do sistema em resposta ao evento `runlevel` e ao novo runlevel.

O processo pode se tornar complicado, pois a origem dos eventos nem sempre é clara.

O Upstart gera somente alguns eventos, e o restante é proveniente dos jobs. Os arquivos de configuração dos jobs normalmente declaram os eventos que serão gerados, porém os detalhes de como os jobs os geram não estão, em geral, nos arquivos de configuração de jobs do Upstart.

Para chegar aos detalhes, com frequência, será preciso explorar mais a fundo. Por exemplo, considere o evento `static-network-up`. O arquivo de configuração do job *network-interface.conf* diz que ele gera esse evento, porém não diz em que ponto. O fato é que o evento é originado pelo comando `ifup`, que esse job executa quando inicializa uma interface de rede no script `/etc/network/if-up.d/upstart`.

👍 **Observação:** embora tudo isso esteja documentado [o diretório *ifup.d* está na página de manual `interfaces(5)`, referenciado na página de manual `ifup(8)`], descobrir como tudo isso funciona somente lendo a documentação pode ser um desafio. Geralmente, é mais rápido executar `grep` com o nome do evento em vários arquivos de configuração para ver o que aparece e, em seguida, tentar consolidar tudo a partir daí.

Um problema com o Upstart é que, atualmente, não há nenhuma maneira clara de ver os eventos. Você pode mudar a prioridade de log para debug, o que fará com que seja feito log de tudo que ocorrer (normalmente em `/var/log/syslog`), porém a quantidade enorme de informações extras nesse arquivo faz com que seja difícil determinar o contexto de um evento.

6.5.2 Jobs do Upstart

Cada arquivo do diretório de configuração `/etc/init` do Upstart corresponde a um job, e o arquivo de configuração principal para cada job tem uma extensão `.conf`. Por exemplo, `/etc/init/mountall.conf` define o job *mountall*.

Há dois tipos principais de jobs no Upstart.

- Jobs de tarefa (task jobs) – são jobs com um término claro. Por exemplo, *mountall* é um job de tarefa porque ele termina quando acaba de montar os sistemas de arquivos.
- Jobs de serviço (service jobs) – esses jobs não têm um término definido. Os servidores (daemons) como o `udev`, os servidores de banco de dados e os servidores web são todos jobs de serviço.

Um terceiro tipo de job é o *job abstrato*. Pense nele como um tipo de job de serviço virtual. Os jobs abstratos existem somente no Upstart e não iniciam nada por si só, porém, às vezes, são usados como ferramentas de administração de outros jobs porque esses podem iniciar e terminar de acordo com os eventos provenientes de um job abstrato.

Visualizando os jobs

Você pode visualizar os jobs do Upstart e seus status usando o comando `initctl`. Para ter uma visão geral do que está acontecendo em seu sistema, execute:

```
$ initctl list
```

Haverá muitos dados de saída, portanto vamos dar uma olhada somente em dois jobs de exemplo que poderão aparecer em uma listagem típica. Eis um exemplo simples do status de um job de tarefa:

```
mountall stop/waiting
```

Essa saída indica que o job de tarefa *mountall* tem um status igual a `stop/waiting` (parar/em espera), que significa que ele não está em execução. (Infelizmente, na época desta publicação, não era possível usar o status para determinar se um job já havia executado ou não, pois `stop/waiting` também se aplicava a jobs que nunca executaram.)

Os jobs de serviço que têm processos associados aparecem na listagem de status do seguinte modo:

```
tty1 start/running, process 1634
```

Essa linha mostra que o job *tty1* está executando e que o processo de ID 1634 está executando o job. (Nem todos os jobs de serviço têm processos associados.)

👍 **Observação:** se souber o nome de um job, você poderá ver o seu status diretamente, usando `initctl status job`.

A parte referente ao status da saída de `initctl` (por exemplo, `stop/waiting`) pode ser confusa. O lado esquerdo (antes da `/`) é o *objetivo* (goal), ou em direção a que supõe-se que o job está trabalhando, por exemplo, iniciar (`start`) ou terminar (`stop`). O lado direito é o *estado corrente do job*, ou o que o job está fazendo nesse momento, por exemplo, esperando (`waiting`) ou executando (`running`). Por exemplo, na listagem anterior, o job *tty1* tem status igual a `start/running`, que significa que seu objetivo é iniciar. O estado `running` indica que ele iniciou com sucesso. (Para jobs de serviço, o estado `running` é nominal.)

O caso *mountall* é um pouco diferente porque os jobs de tarefa não permanecem em execução. O status `stop/waiting` normalmente indica que o job iniciou e completou sua tarefa. Ao concluir sua tarefa, o job passou de um objetivo igual a `start` (iniciar) para `stop` (parar) e agora está esperando mais comandos do Upstart.

Infelizmente, conforme mencionamos antes, como os jobs que jamais iniciaram também têm um status igual a `stop/waiting` no Upstart, não é possível realmente dizer se um job executou ou jamais iniciou, a menos que você habilite o debugging e observe os logs,

conforme descrito na seção 6.5.5.

 **Observação:** você não verá jobs executando em seu sistema que tenham sido iniciados com o recurso de compatibilidade com System V do Upstart.

Transições de estado de jobs

Há vários estados de jobs, porém há uma maneira definida de passar de um estado para outro. Por exemplo, a seguir, apresentamos o modo como um job típico inicia:

1. Todos os jobs começam no status stop/waiting.
2. Quando um usuário ou um evento do sistema inicia um job, o objetivo do job muda de stop para start.
3. O Upstart muda o estado do job de waiting para starting, portanto o status passa a ser start/starting.
4. O Upstart gera um evento *starting job*.
5. O job executa o que quer que ele precise fazer no estado starting.
6. O Upstart muda o estado do job de starting para pre-start e gera o evento *pre-start job*.
7. O job passa por vários outros estados até atingir o estado running.
8. O Upstart gera um evento *started job*.

O término de uma tarefa envolve um conjunto semelhante de mudanças de estado e de eventos. [Veja a página de manual [upstart-events\(7\)](#) para obter detalhes sobre todos os estados e transições para ambos os objetivos (goals).]

6.5.3 Configuração do Upstart

Vamos analisar dois arquivos de configuração: um do job de tarefa *mountall* e outro do job de serviço *tty1*. Assim como todos os arquivos de configuração do Upstart, esse arquivos estão em */etc/init* e se chamam *mountall.conf* e *tty1.conf*. Os arquivos de configuração estão organizados em partes menores chamadas *stanzas*. Cada stanza é iniciada com uma palavra-chave inicial, por exemplo, *description* OU *start*.

Para começar, abra o arquivo *mountall.conf* em seu sistema. Procure uma linha como esta na primeira stanza:

```
Description    "Mount filesystems on boot"
```

Essa stanza oferece um breve texto de descrição do job.

A seguir, você verá algumas stanzas que descrevem como o job *mountall* é iniciado:

```
start on startup
stop on starting rcS
```

Nesse caso, a primeira linha diz ao Upstart para iniciar o job ao receber o evento `startup` (o evento inicial gerado pelo Upstart). A segunda linha diz ao Upstart para terminar o job ao receber o evento `rcS`, quando o sistema vai para modo monousuário (single-user mode).

As duas próximas linhas dizem ao Upstart como o job *mountall* se comporta:

```
expect daemon
task
```

A stanza `task` diz ao Upstart que esse é um job de tarefa, portanto o job deverá ser concluído em algum ponto. A stanza `expect` é ardilosa. Ela significa que o job *mountall* fará spawn de um daemon que irá operar de modo independente do script de job original. O Upstart deve ter essa informação porque deverá saber quando o daemon termina para poder sinalizar corretamente que o job *mountall* terminou. (Discutiremos isso com mais detalhes na seção “Monitoração de processos e a stanza `expect` do Upstart”.)

O arquivo *mountall.conf* prossegue com várias stanzas `emits`, indicando os eventos que os jobs geram:

```
emits virtual-filesystems
emits local-filesystems
emits remote-filesystems
emits all-swaps
emits filesystem
emits mounting
emits mounted
```

 **Observação:** conforme mencionado na seção 6.5.1, apesar de essas linhas estarem presentes, essa não é a verdadeira origem dos eventos. Você deverá vasculhar o script do job para encontrá-la.

Uma stanza `console` também poderá ser vista, indicando o local para onde o Upstart deverá enviar a saída:

```
console output
```

Com o parâmetro `output`, o Upstart envia a saída do job *mountall* para o console do sistema.

Agora você verá os detalhes do job propriamente dito – nesse caso, com uma stanza `script`:

```
script
. /etc/default/rcS
[ -f /forcefsck ] && force_fsck="--force-fsck"
[ "$FSCKFIX" = "yes" ] && fsck_fix="-fsck-fix"
```

```
# set $LANG so that messages appearing in plymouth are translated
if [ -r /etc/default/locale ]; then
    . /etc/default/locale
    export LANG LANGUAGE LC_MESSAGES LC_ALL
fi

exec mountall --daemon $force_fsck $fsck_fix
end script
```

Esse é um shell script (veja o capítulo 11) e, em sua maior parte, é preparatório – define a localidade e se `fsck` é necessário. O comando `exec mountall`, próximo ao final desse script, é o local em que a verdadeira ação acontece. Esse comando monta os sistemas de arquivos e gera os eventos do job quando termina.

Um job de serviço: o `tty1`

O job de serviço `tty1` é muito mais simples; ele controla o prompt de login de um console virtual. Seu arquivo de configuração completo – `tty1.conf` – tem o seguinte aspecto:

```
start on stopped rc RUNLEVEL=[2345] and (
    not-container or
    container CONTAINER=lxc or
    container CONTAINER=lxc-libvirt)

stop on runlevel [!2345]

respawn
exec /sbin/getty -8 38400 tty1
```

A parte mais complicada desse job é, na verdade, quando ele inicia, mas, por enquanto, ignore as linhas `container` e concentre-se nesta parte:

```
start on stopped rc RUNLEVEL=[2345]
```

Essa parte diz ao Upstart para ativar o job ao receber um evento `stopped rc` do Upstart quando o job de tarefa `rc` tiver executado e terminado. Para tornar a condição verdadeira, o job `rc` também deve definir a variável de ambiente `RUNLEVEL` com um valor de 2 a 5 (veja a seção 6.5.6).

 **Observação:** outros jobs que iniciam em runlevels não são tão exigentes. Por exemplo, você poderá ver isto no lugar da linha anterior:

```
start on runlevel [2345]
```

 A única diferença entre essas duas últimas stanzas `start` é o tempo; nesse exemplo, o job é ativado assim que o `runlevel` é definido, enquanto, no exemplo anterior, espera-se até que as atividades do System V terminem.

A configuração de contêiner está presente porque o Upstart não só executa diretamente sobre o kernel do Linux em hardware de verdade, mas também pode executar em

ambientes virtuais ou contêineres. Alguns desses ambientes não têm consoles virtuais, e você não vai querer executar `getty` em um console que não exista.

Terminar o job `tty1` é simples:

```
stop on runlevel [!2345]
```

Essa stanza `stop` diz ao Upstart para terminar o job sempre que o runlevel não estiver entre 2 e 5 (por exemplo, durante o desligamento do sistema).

A stanza `exec` na parte inferior é o comando a ser executado:

```
exec /sbin/getty -8 38400 tty1
```

Essa stanza é muito semelhante à stanza `script` que vimos no job *mountall*, exceto pelo fato de o job `tty1` não ter nenhuma configuração complicada para executar – ela é fácil de ser iniciada com uma única linha. Nesse caso, estamos executando o programa de prompt de login `getty` em `/dev/tty1`, que é o primeiro console virtual (aquele que você obtém quando tecla Ctrl-Alt-F1 em modo gráfico).

A stanza `respawn` instrui o Upstart a reiniciar o job `tty1` se ele terminar. Nesse caso, o Upstart executa um novo `getty` para um novo prompt de login quando você fizer logout do console virtual.

Essa foi a descrição do básico sobre a configuração do Upstart. Você encontrará muito mais detalhes na página de manual `init(5)` e em recursos online, porém há uma stanza que exige atenção especial. A stanza `expect` será discutida a seguir.

Monitoração de processos e a stanza `expect` do Upstart

Como o Upstart monitora processos em jobs depois que esses forem iniciados (para que ele possa terminá-los e reiniciá-los de maneira eficiente), ele quer saber quais processos são relevantes para cada job. Essa pode ser uma tarefa difícil, pois, no esquema tradicional de inicialização do Unix, os processos fazem `fork` de outros processos durante a inicialização para se transformarem em daemons, e o processo principal de um job pode iniciar após um ou dois `forks`. Sem uma monitoração adequada dos processos, o Upstart não poderá finalizar a inicialização de seus jobs, ou irá monitorar o PID incorreto para o job.

A maneira como um job se comporta é informada ao Upstart por meio da stanza `expect`. Há quatro possibilidades básicas:

- Sem stanza `expect` – o processo principal do job não realiza `fork`. Monitora o processo principal.
- `expect fork` – o processo realiza `fork` uma vez. Monitora o processo gerado com `fork`.

- `expect daemon` — o processo realiza `fork` duas vezes. Monitora o segundo `fork`.
- `expect stop` — o processo principal do job irá gerar um sinal `SIGSTOP` para indicar que está pronto. (Isso é raro.)

Para o Upstart e outras versões modernas de `init`, como o `systemd`, o caso ideal é o primeiro (não há stanza `expect`), pois o processo principal do job não precisará incluir nenhum de seus próprios sistemas de inicialização e de desligamento. Em outras palavras, ele não precisa se preocupar com `forking` ou em se desassociar de um terminal corrente — aborrecimentos com os quais os desenvolvedores de sistemas Unix tiveram de lidar durante anos.

Muitos daemons tradicionais de serviço já incluem opções de estilo de debugging que informam ao processo principal para não efetuar `fork`. Exemplos incluem o daemon Secure SHell — `sshd` — e sua opção `-D`. Uma olhada nas stanzas de inicialização de `/etc/init/ssh.conf` revela uma configuração simples para iniciar o `sshd`, que evita um `respawning` rápido e elimina uma saída espúria para `stderr`:

```
respawn
respawn limit 10 5
umask 022

# 'sshd -D' leaks stderr and confuses things in conjunction with 'console log'
console none
--trecho omitido--
exec /usr/sbin/sshd -D
```

Entre os jobs que exigem uma stanza `expect`, `expect fork` é a mais comum. Por exemplo, aqui está a parte referente à inicialização do arquivo `/etc/init/cron.conf`:

```
expect fork
respawn

exec cron
```

Uma inicialização simples de job como essa normalmente indica um daemon estável e bem comportado.

 **Observação:** vale a pena ler mais sobre a stanza `expect` no site upstart.ubuntu.com, pois ela se relaciona diretamente com a vida do processo. Por exemplo, você pode rastrear a vida de um processo e suas chamadas de sistema, incluindo `fork()`, por meio do comando `strace`.

6.5.4 Operação do Upstart

Além dos comandos `list` e `status` descritos na seção 6.5.2, o utilitário `initctl` também pode ser usado para controlar o Upstart e seus jobs. Leia a página de manual `initctl(8)` em algum momento, mas, por enquanto, vamos dar uma olhada nos pontos essenciais.

Para iniciar um job Upstart, utilize `initctl start`:

```
# initctl start job
```

Para parar um job, use `initctl stop`:

```
# initctl stop job
```

Para reiniciar um job, use:

```
# initctl restart job
```

Se houver necessidade de gerar um evento para o Upstart, isso pode ser feito manualmente com:

```
# initctl emit evento
```

Você também pode adicionar variáveis de ambiente ao evento gerado ao acrescentar parâmetros *chave=valor* após *evento*.

 **Observação:** você não pode iniciar e parar serviços individuais que foram iniciados com o recurso de compatibilidade com System V do Upstart. Veja a seção 6.6.1 para obter mais informações sobre como fazer isso em um script System V `init`.

Há várias maneiras de desabilitar um job Upstart para que ele não seja iniciado no momento do boot, porém o modo mais fácil de ser mantido é determinar o nome do arquivo de configuração do job (normalmente, é `/etc/init/<job>.conf`) e, em seguida, criar um novo arquivo chamado `/etc/init/<job>.override` contendo somente a linha:

```
manual
```

Agora a única maneira de o job iniciar é executando `initctl start job`.

A principal vantagem desse método é que ele é facilmente reversível. Para permitir que o job seja iniciado novamente no boot, remova o arquivo `.override`.

6.5.5 Logs do Upstart

Há dois tipos básicos de log no Upstart: logs de jobs de serviço e mensagens de diagnóstico que o próprio Upstart gera. Os logs de jobs de serviço registram a saída-padrão e o erro-padrão dos scripts e dos daemons que executam os serviços. Essas mensagens, que são registradas em `/var/log/upstart`, são adicionais às mensagens-padrão do syslog que um serviço pode gerar. (Você aprenderá mais sobre o syslog no capítulo 7.) É difícil classificar o que é enviado a esses logs porque não há nenhum padrão, mas o conteúdo mais comum refere-se a mensagens de inicialização e de desligamento bem como mensagens de erro de emergência. Muitos serviços não geram nenhuma mensagem porque eles enviam tudo para o syslog ou para o seu próprio recurso de logging.

O próprio log de diagnósticos do Upstart pode conter informações sobre quando ele é iniciado e recarregado, assim como determinadas informações sobre jobs e eventos. Esse log de diagnósticos é enviado para o utilitário `syslog` do kernel. No Ubuntu, normalmente, você encontrará esse log no arquivo `/var/log/kern.log` e no arquivo `/var/log/syslog`, que armazena de tudo.

Apesar do que foi dito, por padrão, o Upstart efetua pouco ou quase nenhum log, portanto, para ver algo nos logs, é preciso alterar a prioridade de log do Upstart. O nome da prioridade default é `message`. Para efetuar o log de eventos e de mudanças de jobs em um sistema em execução, altere a prioridade do log para `info`:

```
# initctl log-priority info
```

Tenha em mente que isso não será permanente e que a prioridade será restaurada após uma reinicialização do sistema. Para fazer o Upstart efetuar o log de tudo quando ele for iniciado, adicione um parâmetro `--verbose` como parâmetro de boot, conforme descrito na seção 5.5.

6.5.6 Runlevels do Upstart e compatibilidade com System V

Até agora, tocamos em alguns pontos em que o Upstart suporta a ideia de runlevels do System V e mencionamos que ele tem a capacidade de executar scripts de inicialização do System V como um job. A seguir, apresentamos uma visão geral mais detalhada de como isso funciona em sistemas Ubuntu:

1. O job `rc-sysinit` inicia normalmente após receber os eventos `filesystem` e `static-network-up`. Antes de ele executar, não há nenhum runlevel.
2. O job `rc-sysinit` determina em que runlevel deve entrar. Geralmente, o runlevel é o default, porém o job pode efetuar o parse de um arquivo `/etc/inittab` mais antigo, ou pode obter o runlevel a partir de um parâmetro de kernel (em `/proc/cmdline`).
3. O job `rc-sysinit` executa `telinit` para mudar de runlevel. O comando gera um evento `runlevel`, que especifica o runlevel na variável de ambiente `RUNLEVEL`.
4. O Upstart recebe o evento `runlevel`. Vários jobs estão configurados para iniciar com o evento `runlevel`, em conjunto com um determinado runlevel, e o Upstart coloca isso em ação.
5. Um dos jobs de tarefa ativados pelo runlevel – o `rc` – é responsável por executar a inicialização do System V. Para isso, o job `rc` executa `/etc/init.d/rc`, assim como o System V `init` faria (veja a seção 6.6).
6. Depois que o job `rc` terminar, o Upstart poderá iniciar vários outros jobs ao

receber o evento `stopped rc` (por exemplo, o job *tty1* descrito na seção “Um job de serviço: o *tty1*”).

Observe que embora o Upstart não trate o runlevel de modo diferente de qualquer outro evento, muitos dos arquivos de configuração de jobs na maioria dos sistemas Upstart referenciam o runlevel.

Qualquer que seja o caso, há um ponto crítico durante o boot, quando os sistemas de arquivos são montados e a maior parte da inicialização importante do sistema foi feita. Nesse ponto, o sistema estará pronto para iniciar os serviços de sistema de mais alto nível, por exemplo, gerenciadores de display gráfico e servidores de banco de dados. Um evento `runlevel` é prático para marcar esse ponto. No entanto você pode configurar o Upstart para usar qualquer evento como trigger. Um desafio surge ao tentar determinar quais serviços são iniciados como jobs do Upstart e quais iniciam em modo de compatibilidade com System V. A maneira mais fácil de descobrir é dar uma olhada no link `farm` do System V de seu runlevel (veja a seção 6.6.2). Por exemplo, se o seu runlevel for 2, dê uma olhada em */etc/rc2.d*; tudo o que estiver aí provavelmente estará executando em modo de compatibilidade com System V.

 **Observação:** uma pedra no caminho pode ser a presença de scripts dummy em */etc/init.d*. Para qualquer job de serviço do Upstart, pode haver também um script de estilo System V para esse serviço em */etc/init.d*, porém esse script não fará nada além de informá-lo que o serviço foi convertido para um job Upstart. Tampouco haverá um link para o script no diretório de links do System V. Se você se deparar com um script dummy, descubra o nome do job no Upstart e use `initctl` para controlar o job.

6.6 System V init

A implementação do System V init no Linux data dos primórdios do Linux; sua ideia central é suportar um boot organizado do sistema para diferentes runlevels, com uma inicialização de processos cuidadosamente sequenciada. Embora o System V atualmente não seja comum na maioria das instalações desktop, você poderá encontrar o System V init no Red Hat Enterprise Linux bem como em ambientes Linux embarcados, por exemplo, em roteadores e em telefones.

Há dois componentes principais em uma instalação típica do System V init: um arquivo de configuração central e um grande conjunto de scripts de boot expandido por um `farm` de links simbólicos. É no arquivo de configuração */etc/inittab* que tudo começa. Se você tiver o System V init, procure uma linha como a que se segue em seu arquivo *inittab*:

```
id:5:initdefault:
```

Essa linha indica que o runlevel default é 5.

Todas as linhas em *inittab* assumem o formato a seguir, com quatro campos separados por dois-pontos, nesta ordem:

- um identificador único (uma string curta, por exemplo, *id* no exemplo anterior);
- o(s) número(s) aplicável(is) de runlevel;
- a ação que *init* deve executar (definir o runlevel default como 5 no exemplo anterior);
- um comando a ser executado (opcional).

Para ver como os comandos funcionam em um arquivo *inittab*, considere a linha a seguir:

```
l5:5:wait:/etc/rc.d/rc 5
```

Essa linha em particular é importante porque ela dispara a maior parte da configuração do sistema e os serviços. Nesse caso, a ação *wait* determina quando e como o System V *init* executa o comando: executa */etc/rc.d/rc 5* uma vez quando entrar no runlevel 5 e, em seguida, espera esse comando terminar antes de fazer qualquer outra tarefa. Para abreviar uma história longa, o comando *rc 5* executa tudo o que estiver em */etc/rc5.d* que comece com um número (na ordem dos números).

A seguir estão algumas das ações mais comuns em *inittab*, além de *initdefault* e de *wait*.

respawn

A ação *respawn* diz ao *init* para executar o comando que se segue e, se o comando terminar a execução, ele deverá ser executado novamente. É provável que você vá ver algo como o que se segue em um arquivo *inittab*:

```
l:2345:respawn:/sbin/mingetty tty1
```

Os programas *getty* fornecem prompts de login. A linha anterior é usada para o primeiro console virtual (*/dev/tty1*), que é aquele que você vê quando tecla Alt-F1 ou Ctrl-Alt-F1 (veja a seção 3.4.4). A ação *respawn* traz o prompt de login de volta depois que você fizer logout.

ctrlaltdel

A ação *ctrlaltdel* controla o que o sistema faz quando você tecla Ctrl-Alt-Del em um console virtual. Na maioria dos sistemas, esse é um tipo de comando de reinicialização de sistema, que usa o comando *shutdown* (será discutido na seção 6.7).

sysinit

A ação *sysinit* é a primeira ação que *init* deve executar ao iniciar, antes de entrar em qualquer runlevel.

👉 **Observação:** para ver mais ações disponíveis, consulte a página de manual `inittab(5)`.

6.6.1 System V init: sequência de comandos de inicialização

Agora você está pronto para saber como o System V init inicia os serviços do sistema, imediatamente antes de permitir que você faça login. Lembre-se desta linha de *inittab* que vimos anteriormente:

```
l5:5:wait:/etc/rc.d/rc 5
```

Essa pequena linha dispara vários outros programas. Com efeito, `rc` quer dizer *run commands* (executar comandos), aos quais muitas pessoas se referem como scripts, programas ou serviços. Mas onde estão esses comandos?

O 5 nessa linha nos informa que estamos falando do runlevel 5. Os comandos provavelmente estão em `/etc/rc.d/rc5.d` ou em `/etc/rc5.d`. (O runlevel 1 usa *rc1.d*, o runlevel 2 usa *rc2.d* e assim por diante.) Por exemplo, você poderá encontrar os itens a seguir no diretório *rc5.d*:

```
S10sysklogd  S20ppp      S99gpm
S12kerneld  S25netstd_nfs S99httpd
S15netstd_init S30netstd_misc S99rnmologin
S18netbase   S45pcmcia    S99sshd
S20acct      S89atd
S20logoutd   S89cron
```

O comando `rc 5` inicia os programas do diretório *rc5.d* ao executar os comandos a seguir, nesta sequência:

```
S10sysklogd start
S12kerneld start
S15netstd_init start
S18netbase start
--trecho omitido--
S99sshd start
```

Observe o argumento `start` em cada comando. A letra *S* maiúscula em um nome de comando significa que ele deve ser executado em modo *start*, e o número (de 00 a 99) determina em que ponto da sequência o `rc` inicia o comando. Os comandos *rc*.d* geralmente são shell scripts que iniciam programas em */sbin* ou em */usr/sbin*.

Normalmente, você pode descobrir o que um comando em particular faz ao visualizar o script usando `less` ou outro programa que efetue paginação.

👉 **Observação:** alguns diretórios *rc*.d* contêm comandos que começam com *K* (de “kill”, ou modo stop). Nesse caso, o `rc` executa o comando com o argumento `stop` no lugar de `start`. É mais provável que você vá encontrar comandos *K* em runlevels que efetuem o desligamento do sistema.

Esses comandos podem ser executados manualmente. Normalmente, porém, você fará isso por meio do diretório *init.d* no lugar dos diretórios *rc*.d*, que iremos descrever agora.

6.6.2 O link farm do System V init

O conteúdo dos diretórios *rc*.d*, na realidade, são links simbólicos para arquivos em outro diretório, o *init.d*. Se sua meta é interagir com, adicionar, apagar ou modificar serviços nos diretórios *rc*.d*, você deve entender esses links simbólicos. Uma listagem longa de um diretório como *rc5.d* revela uma estrutura como esta:

```
lrwxrwxrwx . . . S10sysklogd -> ../init.d/sysklogd
lrwxrwxrwx . . . S12kernel.d -> ../init.d/kernel.d
lrwxrwxrwx . . . S15netstd_init -> ../init.d/netstd_init
lrwxrwxrwx . . . S18netbase -> ../init.d/netbase
--trecho omitido--
lrwxrwxrwx . . . S99httpd -> ../init.d/httpd
--trecho omitido--
```

Uma grande quantidade de links simbólicos em vários subdiretórios como esse é chamada de *link farm*. As distribuições Linux contêm esses links para que elas possam usar os mesmos scripts de inicialização para todos os runlevels. Essa convenção não é um requisito, porém simplifica a organização.

Iniciando e finalizando serviços

Para iniciar e finalizar serviços manualmente, utilize o script no diretório *init.d*. Por exemplo, uma maneira de iniciar o programa de servidor web *httpd* manualmente é executar *init.d/httpd start*. De modo semelhante, para matar um serviço em execução, o argumento *stop* pode ser usado (*httpd stop*, por exemplo).

Modificando a sequência de boot

A alteração da sequência de boot no System V init normalmente é feita modificando-se o link farm. A alteração mais comum consiste em evitar que um dos comandos no diretório *init.d* execute em um determinado runlevel. Você deve tomar cuidado em relação ao modo como isso é feito. Por exemplo, você pode considerar a remoção do link simbólico no diretório *rc*.d* apropriado. Porém tome cuidado: se, algum dia, for necessário restaurar o link, você poderá ter problemas para se lembrar do seu nome exato. Uma das melhores maneiras de fazer isso é adicionar um underscore (*_*) no início do nome do link, desta maneira:

```
# mv S99httpd _S99httpd
```

Essa mudança faz com que `rc` ignore `_S99httpd`, pois o nome do arquivo não começa mais com `S` ou com `K`, porém o nome original permanece óbvio.

Para adicionar um serviço, crie um script como aqueles que estão no diretório `init.d` e, em seguida, crie um link simbólico no diretório `rc*.d` correto. A maneira mais fácil é copiar e modificar um dos scripts que já estejam em `init.d` e que você entenda (veja o capítulo 11 para obter mais informações sobre shell scripts).

Ao adicionar um serviço, selecione um local apropriado na sequência de boot para iniciá-lo. Se o serviço começar cedo demais, ele poderá não funcionar em virtude de uma dependência de algum outro serviço. Para serviços que não sejam essenciais, a maioria dos administradores de sistema prefere números na casa dos 90, o que coloca os serviços após a maioria dos serviços que vêm com o sistema.

6.6.3 run-parts

O sistema que o System V init usa para executar os scripts de `init.d` encontrou lugar em vários sistemas Linux, independentemente de usarem ou não o System V init. Esse sistema consiste de um utilitário que se chama `run-parts`, e sua única tarefa é executar um conjunto de programas executáveis em um dado diretório, de acordo com algum tipo de ordem previsível. Você pode pensar nele quase como uma pessoa que execute o comando `ls` em algum diretório e, em seguida, simplesmente execute qualquer programa que for visto na saída.

O comportamento-padrão consiste em executar todos os programas em um diretório, porém, com frequência, você terá a opção de selecionar determinados programas e ignorar outros. Em algumas distribuições, não é preciso ter muito controle sobre os programas que serão executados. Por exemplo, o Fedora vem com um utilitário `run-parts` bem simples.

Outras distribuições, como Debian e Ubuntu, têm um programa `run-parts` mais complexo. Seus recursos incluem a capacidade de executar programas de acordo com uma expressão regular (por exemplo, usando a expressão `S[0-9]{2}` para executar todos os scripts “start” em um diretório de runlevel `/etc/init.d`) e passar argumentos para os programas. Essas capacidades permitem iniciar e finalizar runlevels do System V com um único comando.

Você não precisa realmente entender os detalhes de como usar o `run-parts`; com efeito, a maioria das pessoas nem sabe que o `run-parts` existe. Os principais pontos a serem lembrados são o fato de ele aparecer ocasionalmente em scripts e que ele existe somente para executar programas em um dado diretório.

6.6.4 Controlando o System V init

Ocasionalmente, você deverá dar um pequeno sinal ao init para lhe dizer para mudar de runlevel, reler sua configuração ou desligar o sistema. Para controlar o System V init, utilize o `telinit`. Por exemplo, para mudar para o runlevel 3, digite:

```
# telinit 3
```

Ao mudar de runlevel, o init tenta matar qualquer processo que não esteja no arquivo *inittab* do novo runlevel, portanto tome cuidado quando fizer isso.

Quando for necessário adicionar ou remover jobs, ou fazer qualquer outra mudança no arquivo *inittab*, você deve informar o init a respeito da mudança e fazer com que ele recarregue o arquivo. O comando `telinit` para isso é:

```
# telinit q
```

`telinit s` também pode ser usado para mudar para o modo monousuário (veja a seção 6.9).

6.7 Desligando o seu sistema

O init controla como o sistema é desligado e reiniciado. Os comandos para desligar o sistema são os mesmos, independentemente de qual versão de init estiver sendo executada. A maneira apropriada de desligar um computador Linux é usando o comando `shutdown`.

Há duas maneiras básicas de usar `shutdown`. Se você fizer um *halt* no sistema, o computador será desligado e será mantido desligado. Para efetuar um halt imediato do computador, execute:

```
# shutdown -h now
```

Na maioria dos computadores e versões de Linux, um halt interrompe a alimentação do computador. Você também pode efetuar uma *reinicialização* (reboot) do computador. Para uma reinicialização, utilize `-r` no lugar de `-h`.

O processo de desligamento dura vários segundos. Você não deve efetuar um reset nem desligar um computador durante esse estágio.

No exemplo anterior, `now` é o horário para efetuar o desligamento. Esse argumento é obrigatório, porém existem várias maneiras de especificar o horário. Por exemplo, se você quiser que o computador desligue em algum momento no futuro, use `+n`, em que *n* é o número de minutos que `shutdown` deve esperar antes de realizar o seu trabalho. [Para ver outras opções, consulte a página de manual `shutdown(8)`].

Para fazer o sistema reiniciar em dez minutos, digite:

shutdown -r +10

No Linux, `shutdown` notifica qualquer um que estiver logado, informando que o computador será desligado, porém muito pouco trabalho de verdade é feito. Se você especificar um horário diferente de `now`, o comando `shutdown` criará um arquivo chamado `/etc/nologin`. Quando esse arquivo estiver presente, o sistema proibirá logins de qualquer pessoa, exceto do superusuário.

Quando o horário de desligamento do sistema finalmente for atingido, `shutdown` dirá ao `init` para iniciar o processo de desligamento. No `systemd`, isso significa ativar as unidades de desligamento; no `Upstart`, significa gerar eventos de desligamento, e no `System V init`, significa mudar o `runlevel` para 0 ou 6. Independentemente da implementação ou da configuração do `init`, o procedimento, em geral, ocorre da seguinte maneira:

1. O `init` pede que cada processo seja encerrado de forma limpa.
2. Se um processo não responder após um período de tempo, o `init` o matará, inicialmente tentando usar um sinal `TERM`.
3. Se o sinal `TERM` não funcionar, o `init` usará o sinal `KILL` em qualquer retardatário.
4. O sistema bloqueará os arquivos de sistema e fará outras preparações para o desligamento.
5. O sistema desmontará todos os sistemas de arquivos que não sejam o sistema-raiz.
6. O sistema remonta o sistema de arquivos-raiz somente para leitura.
7. O sistema escreve todos os dados em buffer no sistema de arquivos usando o programa `sync`.
8. O último passo consiste em dizer ao kernel para reiniciar ou parar por meio da chamada de sistema `reboot(2)`. Isso pode ser feito pelo `init` ou por um programa auxiliar como `reboot`, `halt` ou `poweroff`.

Os programas `reboot` e `halt` se comportam de modo diferente, de acordo com o modo como são chamados, o que pode causar confusão. Por padrão, esses programas chamam `shutdown` com as opções `-r` ou `-h`. Entretanto, se o sistema já estiver em um `runlevel` para `halt` ou `reboot`, os programas dirão ao kernel para que desligue imediatamente. Se você realmente quiser desligar o seu computador apressadamente, não se importando com qualquer dano em potencial causado por um desligamento desordenado, utilize a opção `-f` (force).

6.8 O sistema de arquivos inicial em RAM

O processo de boot do Linux, em sua maior parte, é bem simples. Entretanto, de certa forma, um componente sempre causa confusão: o *initramfs*, ou *sistema de arquivos inicial em RAM* (initial RAM filesystem). Pense nele como uma pequena porção do espaço de usuário que fica na frente da inicialização normal do modo usuário. Inicialmente, porém, vamos falar do porquê de sua existência.

O problema tem origem na disponibilidade de vários tipos diferentes de hardware de armazenamento. Lembre-se de que o kernel do Linux não conversa com as interfaces do PC BIOS ou do EFI para obter dados dos discos, portanto, para montar seu sistema de arquivos raiz, ele precisa de suporte de drivers para o sistema de armazenamento subjacente. Por exemplo, se a raiz estiver em um array RAID conectado a um controlador de terceiros, o kernel precisará do driver desse controlador antes. Infelizmente, há tantos drivers de controladores de armazenamento que as distribuições não conseguem incluir todos eles em seus kernels, portanto muitos drivers são disponibilizados como módulos carregáveis. Porém os módulos carregáveis são arquivos e se o seu kernel não tiver um sistema de arquivos montado anteriormente, ele não poderá carregar os módulos de drivers necessários.

A solução alternativa consiste em reunir uma pequena coleção de módulos de drivers de kernel, juntamente com alguns utilitários, em um arquivo. O boot loader carrega esse arquivo para a memória antes de executar o kernel. Na inicialização, o kernel lê o conteúdo do arquivo em um sistema de arquivos temporário em RAM (o *initramfs*), monta-o em `/` e efetua a passagem para o modo usuário ao `init` no *initramfs*. Em seguida, os utilitários incluídos no *initramfs* permitem ao kernel carregar os módulos de driver necessários para o verdadeiro sistema de arquivos-raiz. Por fim, os utilitários montam o verdadeiro sistema de arquivos-raiz e iniciam o verdadeiro `init`.

As implementações variam e estão em constante evolução. Em algumas distribuições, o `init` no *initramfs* é um shell script bem simples que inicia um `udev` para carregar os drivers e depois monta a verdadeira raiz e executa o `init` a partir daí. Em distribuições que usam o `systemd`, normalmente, você verá toda uma instalação `systemd` ali, sem arquivos de configuração de unidade e somente com alguns arquivos de configuração do `udev`.

Uma característica básica do sistema de arquivos inicial em RAM que permaneceu inalterada (até agora) desde a sua origem é a capacidade de ele ser ignorado caso não seja necessário. Isso quer dizer que se o seu kernel tiver todos os drivers de que precisar para montar o seu sistema de arquivos-raiz, você poderá omitir o sistema de arquivos inicial em RAM na configuração de seu boot loader. Quando for bem-sucedido, eliminar o sistema de arquivos inicial em RAM reduz o tempo de boot,

normalmente em alguns segundos. Tente fazer isso por conta própria no momento do boot usando o editor de menu do GRUB para remover a linha `initrd`. (É melhor não efetuar experimentos mudando o arquivo de configuração do GRUB, pois você poderá cometer um erro que será difícil de corrigir.) Recentemente, tem sido um pouco mais difícil ignorar o sistema de arquivos inicial em RAM porque recursos como montagem baseada em UUID podem não estar disponíveis em kernels de distribuições genéricas.

É fácil ver o conteúdo de seu sistema de arquivos inicial em RAM porque, na maioria dos sistemas modernos, ele é formado simplesmente por arquivos `cpio` compactados com `gzip` [veja a página de manual `cpio(1)`]. Inicialmente, encontre o arquivo compactado dando uma olhada na configuração de seu boot loader (por exemplo, execute `grep` em busca de linhas com `initrd` em seu arquivo de configuração `grub.cfg`). Em seguida, use `cpio` para efetuar o dump do conteúdo do arquivo em um diretório temporário em algum local e examine o resultado. Por exemplo:

```
$ mkdir /tmp/myinitrd
$ cd /tmp/myinitrd
$ zcat /boot/initrd.img-3.2.0-34 | cpio -i --no-absolute-file-names
--trecho omitido--
```

Uma parte especialmente interessante é o “pivô” próximo ao final do processo `init` no sistema de arquivos inicial em RAM. Essa parte é responsável por remover o conteúdo do sistema de arquivos temporário (para economizar memória) e efetuar a mudança de forma permanente para a verdadeira raiz.

Normalmente, você não criará o seu próprio sistema de arquivos inicial em RAM, pois esse é um processo complicado. Há vários utilitários para criar imagens de sistema de arquivos iniciais em RAM, e é bem provável que a sua distribuição venha com um deles. Dois dos utilitários mais comuns são `dracut` e `mkinitramfs`.

👍 **Observação:** o termo *sistema de arquivos inicial em RAM* (`initramfs`) refere-se à implementação que utiliza o arquivo `cpio` como fonte para o sistema de arquivos temporário. Há uma versão mais antiga chamada disco inicial em RAM – ou `initrd` (initial RAM disk) – que utiliza uma imagem de disco como base para o sistema de arquivos temporário. Isso caiu em desuso porque é muito mais fácil manter um arquivo `cpio`. No entanto, com frequência, você verá o termo *initrd* sendo usado para se referir a um sistema de arquivos inicial em RAM baseado em `cpio`. Geralmente, como no exemplo anterior, os nomes de arquivo e os arquivos de configuração ainda conterão `initrd`.

6.9 Inicialização de emergência e modo monousuário

Quando algo dá errado no sistema, o primeiro recurso geralmente é inicializar o sistema com uma “live image” de uma distribuição (as imagens de instalação da maioria das distribuições também servem como live images) ou com uma imagem

dedicada de resgate como SystemRescueCd, que poderá ser inserida em uma mídia removível. Tarefas comuns para corrigir um sistema incluem:

- efetuar verificações em sistemas de arquivos após uma falha do sistema;
- reconfigurar uma senha de root que tenha sido esquecida;
- corrigir problemas em arquivos críticos, por exemplo, em */etc/fstab* e em */etc/passwd*;
- fazer restaurações a partir de backups após uma falha do sistema.

Outra opção para efetuar rapidamente uma inicialização e atingir um estado funcional é usar o *modo monousuário* (single-user mode). A ideia é que o sistema faça boot rapidamente para um root shell, em vez de passar por toda a confusão de serviços. No System V init, o modo monousuário normalmente corresponde ao runlevel 1, e você também pode entrar nesse modo com um parâmetro *-s* no boot loader. Pode ser necessário fornecer a senha de root para entrar em modo monousuário.

O maior problema com o modo monousuário é que ele não oferece muitas amenidades. É quase certo que a rede não estará disponível (e se estiver, será difícil usá-la), você não terá uma GUI e o seu terminal poderá nem mesmo funcionar corretamente. Por esse motivo, as live images quase sempre são preferíveis.

CAPÍTULO 7

Configuração do sistema: logging, hora do sistema, tarefas em lote e usuários

Ao observar o diretório */etc* pela primeira vez, você poderá se sentir um pouco oprimido. Embora a maioria dos arquivos que você vir afetem as operações de um sistema em certa medida, apenas alguns são fundamentais.

O assunto deste capítulo inclui as partes do sistema que tornam a infraestrutura discutida no capítulo 4 disponível às ferramentas de nível de usuário discutidas no capítulo 2. Em particular, iremos dar uma olhada no seguinte:

- nos arquivos de configuração que as bibliotecas de sistema acessam para obter informações de servidores e de usuários;
- nos programas servidores (às vezes chamados de *daemons*) que executam quando o sistema é inicializado;
- nos utilitários de configuração que podem ser usados para ajustar os programas servidores e os arquivos de configuração;
- nos utilitários de administração.

Como nos capítulos anteriores, não há praticamente nenhum material sobre rede aqui, pois a rede é um bloco de construção à parte do sistema. No capítulo 9, você verá em que local a rede se encaixa.

7.1 Estrutura do */etc*

A maioria dos arquivos de configuração do sistema em um sistema Linux encontra-se em */etc*. Historicamente, cada programa tinha um ou mais arquivos de configuração nesse local e, pelo fato de haver muitos pacotes em um sistema Unix, */etc* acumulava arquivos rapidamente.

Havia dois problemas com essa abordagem: era difícil encontrar arquivos de configuração particulares em um sistema em execução e era difícil manter um sistema configurado dessa maneira. Por exemplo, se você quisesse alterar a configuração do logger do sistema, seria necessário editar */etc/syslog.conf*. Porém, após ter feito a sua

alteração, um upgrade de sua distribuição poderia limpar suas personalizações.

Há vários anos, a tendência tem sido colocar os arquivos de configuração do sistema em subdiretórios de */etc*, como você já viu no caso dos diretórios de boot (*/etc/init* para o Upstart e */etc/systemd* para o systemd). Ainda há alguns arquivos de configuração individuais em */etc*, porém, na maioria das vezes, se o comando `ls -F /etc` for executado, você verá que a maior parte dos itens nesse local atualmente são subdiretórios.

Para solucionar o problema de sobrescrever os arquivos de configuração, agora você pode colocar suas personalizações em arquivos separados nos subdiretórios de configuração, como aqueles em */etc/grub.d*.

Que tipo de arquivo de configuração é encontrado em */etc*? A diretriz básica é que as configurações personalizáveis para um único computador, como informações de usuários (*/etc/passwd*) e detalhes de rede (*/etc/network*), sejam colocadas em */etc*. Contudo detalhes de aplicações gerais, como os defaults de uma distribuição para uma interface de usuário, não pertencem a */etc*. E, com frequência, você perceberá que arquivos de configuração não personalizáveis do sistema poderão ser encontrados em outros locais, como ocorre com os arquivos de unidade prontos do systemd em */usr/lib/systemd*.

Você já viu alguns dos arquivos de configuração pertinentes à inicialização do sistema. Agora daremos uma olhada em um serviço de sistema típico e veremos como podemos visualizar e especificar a sua configuração.

7.2 Logging do sistema

A maioria dos programas de sistema escreve suas saídas contendo diagnósticos no serviço *syslog*. O daemon *syslogd* tradicional espera as mensagens e, de acordo com o tipo de mensagem recebida, direciona a saída para um arquivo, a tela, aos usuários ou a alguma combinação deles, ou simplesmente a ignora.

7.2.1 Logger do sistema

O logger do sistema é uma de suas partes mais importantes. Se algo der errado e você não souber por onde deve começar, verifique os arquivos de log do sistema antes. Aqui está um exemplo de uma mensagem do arquivo de log:

```
Aug 19 17:59:48 duplex sshd[484]: Server listening on 0.0.0.0 port 22.
```

A maioria das distribuições Linux executa uma nova versão de *syslogd* chamada *rsyslogd* que faz muito mais do que simplesmente escrever mensagens de log em arquivos. Por

exemplo, você pode usá-lo para carregar um módulo e enviar mensagens de log a um banco de dados. Porém, ao começar a trabalhar com logs de sistema, é mais fácil partir dos arquivos de log normalmente armazenados em `/var/log`. Dê uma olhada em alguns arquivos de log – depois de conhecer sua aparência, você estará pronto para descobrir como eles chegaram até lá.

Muitos dos arquivos em `/var/log` não são mantidos pelo logger do sistema. A única maneira de saber ao certo quais pertencem a `rsyslogd` é dando uma olhada em seu arquivo de configuração.

7.2.2 Arquivos de configuração

O arquivo de configuração base de `rsyslogd` é `/etc/rsyslog.conf`, porém você encontrará determinadas configurações em outros diretórios, por exemplo, em `/etc/rsyslog.d`. O formato da configuração é uma mistura de regras tradicionais e de extensões específicas do `rsyslog`. Uma regra geral é que tudo que começar com um sinal de dólar (\$) é uma extensão.

Uma regra tradicional tem um *seletor* e uma *ação* para mostrar como capturar logs e para onde enviá-los, respectivamente. Por exemplo:

Listagem 7.1 – Regras do syslog

kern.*	/dev/console
*.info;authpriv.none❶	/var/log/messages
authpriv.*	/var/log/secure,root
mail.*	/var/log/maillog
cron.*	/var/log/cron
*.emerg	*❷
local7.*	/var/log/boot.log

O seletor está à esquerda. É o tipo de informação a ser registrada. A lista à direita corresponde à ação: para onde o log deve ser enviado. A maioria das ações da listagem 7.1 compõe-se de arquivos normais, com algumas exceções. Por exemplo, `/dev/console` refere-se a um dispositivo especial para o console do sistema, `root` significa enviar uma mensagem para o superusuário se esse usuário estiver logado e `*` quer dizer enviar mensagens a todos os usuários presentes no sistema no momento. Você também pode enviar mensagens para outro host da rede usando `@host`.

Facilidade e prioridade

O seletor é um padrão que efetua a correspondência de *facilidade* (facility) e de *prioridade* (priority) das mensagens de log. A facilidade é uma classificação geral da

mensagem. (Veja `rsyslog.conf(5)` para obter uma lista de todas as facilidades.)

A função da maioria das facilidades será razoavelmente óbvia a partir de seus nomes. Por exemplo, o arquivo de configuração da listagem 7.1 captura mensagens que ostentem as facilidades `kern`, `authpriv`, `mail`, `cron` e `local7`. Nessa mesma listagem, o asterisco em ❷ é um caractere curinga que captura saídas relacionadas a todas as facilidades.

A *prioridade* vem depois do ponto (`.`), após a facilidade. A ordem das prioridades, da mais baixa para a mais alta, é: `debug`, `info`, `notice`, `warning`, `err`, `crit`, `alert` OU `emerg`.

👍 **Observação:** para excluir mensagens de log de uma facilidade em `rsyslog.conf`, especifique uma prioridade igual a `none`, conforme mostrado em ❶ na listagem 7.1.

Ao colocar uma prioridade específica em um seletor, o `rsyslogd` enviará mensagens com essa prioridade *e com todas as prioridades mais altas* para o destino nessa linha. Desse modo, na listagem 7.1, o `*.info` para a linha em ❶, na realidade, captura a maior parte das mensagens de log e as coloca em `/var/log/messages` porque `info` corresponde a uma prioridade relativamente baixa.

Sintaxe estendida

Conforme mencionado anteriormente, a sintaxe de `rsyslogd` estende a sintaxe tradicional de `syslogd`. As extensões de configuração são chamadas de *diretivas* e, normalmente, começam com um `$`. Uma das extensões mais comuns permite carregar arquivos de configuração adicionais. Dê uma olhada em seu arquivo `rsyslog.conf` em busca de uma diretiva como a que se segue, que faz com que `rsyslogd` carregue todos os arquivos `.conf` em `/etc/rsyslog.d` na configuração:

```
$IncludeConfig /etc/rsyslog.d/*.conf
```

A maioria das demais diretivas estendidas é razoavelmente autoexplicativa. Por exemplo, as diretivas a seguir lidam com usuários e permissões:

```
$FileOwner syslog
$FileGroup adm
$FileCreateMode 0640
$DirCreateMode 0755
$Umask 0022
```

👍 **Observação:** extensões adicionais de arquivos de configuração de `rsyslogd` definem templates e canais de saída. Se for necessário usá-los, a página de manual `rsyslogd(5)` é bem abrangente, porém a documentação baseada em web é mais completa.

Resolução de problemas

Uma das maneiras mais simples de testar o logger do sistema é enviar uma mensagem

de log manualmente por meio do comando `logger`, como mostrado a seguir:

```
$ logger -p daemon.info something bad just happened
```

Pouquíssima coisa pode dar errado com `rsyslogd`. Os problemas mais comuns ocorrem quando uma configuração não captura uma determinada facilidade ou prioridade, ou quando os arquivos de log enchem suas partições de disco. A maioria das distribuições reduz automaticamente os arquivos em `/var/log` por meio de chamadas automáticas a `logrotate` ou a um utilitário semelhante, porém, caso mensagens em excesso cheguem em um curto período de tempo, você ainda poderá ter o disco cheio ou acabar com uma carga alta no sistema.

 **Observação:** os logs capturados por `rsyslogd` não são os únicos registrados por diversas partes do sistema. Discutimos as mensagens de log de inicialização capturadas pelo `systemd` e pelo `Upstart` no capítulo 6, porém você encontrará várias outras fontes, como o servidor `Apache Web`, que normalmente registra seu próprio log de acesso e de erro. Para encontrar esses logs, consulte a configuração do servidor.

Logging: passado e futuro

O serviço `syslog` evoluiu com o passar do tempo. Por exemplo, já houve um `daemon` chamado `klogd` que capturava mensagens de diagnóstico do kernel para o `syslogd`. (Essas mensagens são aquelas que você vê com o comando `dmesg`.) Esse recurso foi incluído no `rsyslogd`.

É quase certo que o logging do sistema Linux irá mudar no futuro. O logging no sistema Unix jamais teve um verdadeiro padrão, porém esforços estão ocorrendo para mudar isso.

7.3 Arquivos para gerenciamento de usuários

Os sistemas Unix permitem ter vários usuários independentes. No nível do kernel, os usuários são simplesmente números (*IDs de usuário*), mas como é muito mais fácil se lembrar de um nome do que de um número, normalmente você trabalhará com *nomes de usuário* (ou *nomes de login*) em vez de usar IDs ao administrar o Linux. Os nomes de usuário existem apenas no espaço de usuário, portanto qualquer programa que trabalhe com um nome de usuário geralmente deve ser capaz de mapear esse nome a um ID de usuário se quiser se referir a um usuário quando estiver conversando com o kernel.

7.3.1 Arquivo `/etc/passwd`

O arquivo `/etc/passwd` em formato texto simples mapeia nomes de usuário a IDs. Ele tem uma aparência semelhante a:

Listagem 7.2 – Uma lista de usuários em `/etc/passwd`

```
root:x:0:0:Superuser:/root:/bin/sh
daemon:*:1:1:daemon:/usr/sbin:/bin/sh
bin:*:2:2:bin:/bin:/bin/sh
sys:*:3:3:sys:/dev:/bin/sh
nobody:*:65534:65534:nobody:/home:/bin/false
juser:x:3119:1000:J. Random User:/home/juser:/bin/bash
beazley:x:143:1000:David Beazley:/home/beazley:/bin/bash
```

Cada linha representa um usuário e tem sete campos separados por dois-pontos. Os campos estão descritos a seguir:

- O nome do usuário.
- A senha criptografada do usuário. Na maioria dos sistemas Linux, a senha não é realmente armazenada no arquivo *passwd*, mas no arquivo *shadow* (veja a seção 7.3.3). O formato do arquivo *shadow* é semelhante ao de *passwd*, porém usuários normais não têm permissão de leitura em *shadow*. O segundo campo em *passwd* ou em *shadow* é a senha criptografada e parece um monte de lixo ilegível, como `d1CvEWiB/oppc`. (As senhas Unix jamais são armazenadas como texto simples.)

Um `x` no segundo campo do arquivo *passwd* indica que a senha criptografada está armazenada no arquivo *shadow*. Um asterisco (*) indica que o usuário não pode fazer login e, se o campo estiver em branco (ou seja, você verá dois dois-pontos seguidos, como em `::`), nenhuma senha será necessária para fazer login. (Tome cuidado com senhas em branco. Jamais tenha um usuário sem senha.)

- O *UID* (user ID, ou ID de usuário), que é a representação do usuário no kernel. Pode haver duas entradas com o mesmo ID de usuário, mas fazer isso irá confundir você, e o seu software também poderá misturá-las. Faça com que o ID de usuário seja único.
- O *GID* (group ID, ou ID de grupo). Essa deve ser uma das entradas numeradas no arquivo `/etc/group`. Os grupos determinam permissões de arquivo e algumas outras características. Esse grupo também é chamado de *grupo primário* do usuário.
- O nome real do usuário (com frequência, chamado de campo *GECOS*). Às vezes, você encontrará vírgulas nesse campo, indicando números de sala e de telefone.
- O diretório home do usuário.
- O shell do usuário (o programa executado quando o usuário iniciar uma sessão de terminal).

A figura 7.1 identifica os vários campos em uma das entradas da listagem 7.2.

Nome de login
Senha
ID do usuário
ID do grupo
Nome real (GECOS)
Diretório home
Shell

juser:x:3119:1000:J. Random User:/home/juser:/bin/bash

Figura 7.1 – Uma entrada do arquivo de senhas.

A sintaxe do arquivo `/etc/passwd` é bem rigorosa e não permite nenhum comentário nem linhas em branco.

👍 **Observação:** um usuário em `/etc/passwd` e um diretório home correspondente, em conjunto, são conhecidos como uma *conta*.

7.3.2 Usuários especiais

Você encontrará alguns usuários especiais em `/etc/passwd`. O *superusuário* (root) sempre tem um UID igual a 0 e um GID igual a 0, como podemos ver na listagem 7.2. Alguns usuários, como *daemon*, não têm privilégios de login. O usuário *nobody* é um usuário desprovido de privilégios. Alguns processos executam como *nobody* porque esse usuário não pode escrever em nada no sistema.

Os usuários que não podem fazer login são chamados de *pseudousuários*. Embora eles não possam fazer login, o sistema pode iniciar processos com seus IDs de usuário. Os pseudousuários como *nobody* geralmente são criados por questões de segurança.

7.3.3 O arquivo `/etc/shadow`

O arquivo de senha *shadow* (`/etc/shadow`) em um sistema Linux normalmente contém informações de autenticação de usuários, incluindo senhas criptografadas e informações sobre vencimento de senhas que correspondem aos usuários em `/etc/passwd`.

O arquivo *shadow* foi introduzido para proporcionar uma maneira mais flexível (e mais segura) de armazenar senhas. Ele incluiu um pacote de bibliotecas e de utilitários, muitos dos quais foram logo substituídos por partes do PAM (veja a seção 7.10). Em vez de introduzir um conjunto totalmente novo de arquivos no Linux, o PAM utiliza `/etc/shadow`, porém não usa determinados arquivos de configuração correspondentes como `/etc/login.defs`.

7.3.4 Manipulando usuários e senhas

Usuários normais interagem com `/etc/passwd` por meio do comando `passwd`. Por padrão,

`passwd` muda a senha do usuário, porém você também pode usar `-f` para mudar o verdadeiro nome do usuário ou `-s` para alterar o shell do usuário para um que esteja listado em `/etc/shells`. (Você também pode usar os comandos `chfn` e `chsh` para alterar o nome verdadeiro e o shell.) O comando `passwd` é um programa `suid-root` porque somente o superusuário pode mudar o arquivo `/etc/passwd`.

Alterando `/etc/passwd` como superusuário

Como `/etc/passwd` usa um formato texto simples, o superusuário pode usar qualquer editor de texto para fazer alterações. Para adicionar um usuário, basta acrescentar uma linha apropriada e criar um diretório `home` para o usuário; para apagar um usuário, faça o inverso. No entanto, para editar o arquivo, é bem provável que você vá querer usar o programa `vipw`, que cria um backup e bloqueia o acesso a `/etc/passwd` enquanto você o estiver editando, como precaução extra. Para editar `etc/shadow` em vez de `/etc/passwd`, utilize `vipw -s`. (Porém é provável que você jamais vá precisar fazer isso.)

A maioria das empresas não gosta de editar `passwd` diretamente porque é muito fácil cometer um erro. É muito mais simples (e mais seguro) fazer alterações nos usuários usando comandos separados disponíveis no terminal ou por meio de GUI. Por exemplo, para definir a senha de um usuário, execute `passwd usuário` como superusuário. Utilize `adduser` e `userdel` para adicionar e remover usuários.

7.3.5 Trabalhando com grupos

Os *grupos* no Unix oferecem uma maneira de compartilhar arquivos com determinados usuários, ao mesmo tempo que impede o acesso a todos os demais. A ideia é que você possa configurar bits para permissão de leitura e de escrita para um grupo em particular, excluindo todos os demais. Essa funcionalidade já foi importante porque muitos usuários compartilhavam um computador, porém ela se tornou menos significativa nos últimos anos à medida que as estações de trabalho são compartilhadas com menos frequência.

O arquivo `/etc/group` define os IDs de grupo (como aqueles encontrados no arquivo `/etc/passwd`). A listagem 7.3 é um exemplo.

Listagem 7.3 – Um arquivo `/etc/group` de exemplo

```
root:*:0:user
daemon:*:1:
bin:*:2:
sys:*:3:
adm:*:4:
```

```
disk:*.6:juser,beazley
nogroup:*.65534:
user:*.1000:
```

Assim como no arquivo */etc/passwd*, cada linha em */etc/group* corresponde a um conjunto de campos separados por dois-pontos. Os campos de cada entrada estão descritos a seguir, da esquerda para a direita.

- Nome do grupo – aparece quando um comando como `ls -l` é executado.
- Senha do grupo – raramente é usada, e você não deverá usá-la (utilize `sudo`, em seu lugar). Use `*` ou qualquer outro valor default.
- ID do grupo (um número) – o GID deve ser único no arquivo *group*. Esse número é usado no campo de grupo de um usuário na entrada de */etc/passwd* para esse usuário.
- Uma lista opcional de usuários que pertencem ao grupo – além dos usuários listados aqui, os usuários com o ID de grupo correspondente nas entradas do arquivo *passwd* também pertencem ao grupo.

A figura 7.2 identifica os campos de uma entrada do arquivo *group*.



Figura 7.2 – Uma entrada do arquivo *group*.

Para ver os grupos aos quais você pertence, execute `groups`.

👍 **Observação:** as distribuições Linux geralmente criam um novo grupo para cada novo usuário adicionado, com o mesmo nome do usuário.

7.4 getty e login

`getty` é um programa que se conecta a terminais e exibe um prompt de login. Na maioria dos sistemas Linux, `getty` não é complicado, pois o sistema o usa somente para logins em terminais virtuais. Em uma listagem de processos, ele normalmente tem uma aparência como a que se segue (por exemplo, ao ser executado em *dev/tty1*):

```
$ ps ao args | grep getty
/sbin/getty 38400 tty1
```

Nesse exemplo, 38400 é o baud rate. Alguns programas `getty` não precisam da

configuração de baud rate. (Os terminais virtuais ignoram o baud rate; ele está presente somente para compatibilidade com versões anteriores com softwares que se conectam com linhas seriais de verdade.)

Após inserir o seu nome de login, o `getty` substitui a si mesmo pelo programa `login`, que solicita sua senha. Se você fornecer a senha correta, `login` substituirá a si mesmo (usando `exec()`) pelo seu shell. Caso contrário, você receberá uma mensagem de “Login incorrect” (login incorreto).

Agora você já sabe o que `getty` e `login` fazem, porém é provável que jamais precisará configurá-los ou alterá-los. De fato, você raramente irá sequer usá-los, pois a maioria dos usuários atualmente faz login por meio de uma interface gráfica como o `gdm` ou remotamente com SSH, e nenhum deles usa `getty` ou `login`. A maior parte do verdadeiro trabalho de autenticação do programa de login é tratada pelo PAM (veja a seção 7.10).

7.5 Configurando o horário

Os computadores Unix dependem de uma manutenção precisa do horário. O kernel mantém o *relógio do sistema* (system clock), que é o relógio consultado quando um comando como `date` é executado. Você também pode configurar o relógio do sistema usando o comando `date`, porém, normalmente, fazer isso não é uma boa ideia porque você nunca terá um horário preciso. O relógio de seu sistema deve estar o mais próximo possível do horário correto.

O hardware do PC tem um *RTC* (Real-Time Clock, ou Relógio de tempo real) com bateria. O RTC não é o melhor relógio do mundo, mas é melhor do que nada. O kernel normalmente configura o seu horário de acordo com o RTC no momento do boot, e você pode reconfigurar o relógio do sistema com o horário atual do hardware usando `hwclock`. Mantenha o seu relógio de hardware em UTC (Universal Coordinated Time) para evitar qualquer problema com fuso horário ou correções referentes a horários de verão. Você pode configurar o RTC de acordo com o relógio UTC de seu kernel usando o comando a seguir:

```
# hwclock --hctosys --utc
```

Infelizmente, o kernel é pior ainda para manter o horário do que o RTC, e como os computadores Unix geralmente ficam ligados durante meses ou anos com um único boot, eles tendem a desenvolver um desvio no horário. *O desvio no horário (time drift) é a diferença atual entre o horário do kernel e o horário verdadeiro (conforme definido por um relógio atômico ou outro relógio bem preciso).*

Você não deve tentar corrigir a diferença com `hwclock`, pois eventos de sistema baseados

em horário poderão ser perdidos ou corrompidos. Um utilitário como `adjtimex` poderá ser executado para atualizar suavemente o relógio, porém, em geral, é melhor manter o horário de seu sistema correto por meio de um daemon de horário de rede (veja a seção 7.5.2).

7.5.1 Representação de horários e fusos horários no kernel

O relógio de sistema do kernel representa o horário corrente como o número de segundos desde meia-noite do dia 1 de janeiro de 1970, UTC. Para ver esse número no instante atual, execute:

```
$ date +%s
```

Para converter esse número em algo que os seres humanos possam ler, os programas do espaço de usuário o alteram para o horário local e efetuam compensações para horários de verão e quaisquer outras circunstâncias anormais (por exemplo, morar em Indiana). O fuso horário local é controlado pelo arquivo `/etc/localtime`. (Não se dê o trabalho de tentar olhar o seu conteúdo; é um arquivo binário.)

Os arquivos de fusos horários de seu sistema estão em `/usr/share/zoneinfo`. Você perceberá que esse diretório contém vários fusos horários e muitos aliases para fusos horários. Para definir manualmente o fuso horário de seu sistema, copie um dos arquivos em `/usr/share/zoneinfo` para `/etc/localtime` (ou crie um link simbólico), ou mude-o usando a ferramenta de fuso horário de sua distribuição. (O programa de linha de comando `tzselect` pode ajudar você a identificar um arquivo de fuso horário.)

Para usar um fuso horário diferente do default do sistema somente para uma sessão de shell, defina a variável de ambiente `TZ` com o nome de um arquivo em `/usr/share/zoneinfo` e teste a alteração desta maneira:

```
$ export TZ=US/Central
$ date
```

Como ocorre com outras variáveis de ambiente, também é possível definir o fuso horário pela duração de um único comando, do seguinte modo:

```
$ TZ=US/Central date
```

7.5.2 Horário da rede

Se o seu computador estiver permanentemente conectado à Internet, você poderá executar um daemon NPT (Network Time Protocol) a fim de manter o horário usando um servidor remoto. Muitas distribuições têm suporte incluído para um daemon NTP, porém ele poderá não estar habilitado por padrão. Talvez seja necessário instalar um

pacote `ntpd` para fazê-lo funcionar.

Se precisar fazer a configuração manualmente, você encontrará ajuda na página web principal do NTP em <http://www.ntp.org/>, porém, se preferir não ler a montanha de documentação ali presente, faça o seguinte:

1. Encontre o servidor NTP mais próximo de seu ISP ou na página web *ntp.org*.
2. Coloque esse servidor de horários em */etc/ntp.conf*.
3. Execute `ntpdate servidor` no momento do boot.
4. Execute `ntpd` no momento do boot, após o comando `ntpdate`.

Se o seu computador não tiver uma conexão permanente com a Internet, você poderá usar um daemon como `chronyd` para manter o horário durante as desconexões.

Seu relógio de hardware também pode ser configurado de acordo com o horário da rede para ajudar o seu sistema a manter a coerência nos horários quando uma reinicialização de sistema for efetuada. (Muitas distribuições fazem isso automaticamente.) Para isso, configure o horário de seu sistema a partir da rede usando `ntpdate` (ou `ntpd`) e, em seguida, execute o comando que vimos anteriormente.

```
# hwclock --systohc --utc
```

7.6 Agendando tarefas recorrentes com cron

O serviço `cron` do Unix executa programas repetidamente com uma agenda fixa. A maioria dos administradores experientes considera o `cron` vital, pois ele pode realizar manutenções automáticas no sistema. Por exemplo, o `cron` executa utilitários de rotação de arquivos de log para garantir que o seu disco rígido não fique cheio com arquivos de log antigos. Aprenda a usar o `cron` porque ele é simplesmente muito útil.

Qualquer programa pode ser executado com o `cron` em qualquer horário que for mais adequado a você. O programa que estiver executando por meio do `cron` é chamado de *cron job*. Para instalar um `cron job`, crie uma linha de entrada em seu *arquivo crontab*, geralmente executando o comando `crontab`. Por exemplo, a entrada do `crontab` a seguir agenda o comando `/home/juser/bin/spmake` para que execute diariamente às 9h15min:

```
15 09 * * * /home/juser/bin/spmake
```

Os cinco campos no início dessa linha, delimitados por espaços em branco, especificam o horário agendado (veja também a figura 7.3). Os campos estão descritos a seguir, na sequência:

- Minuto (de 0 a 59). O `cron job` anterior está definido para o minuto 15.

- Hora (de 0 a 23). O job anterior está definido para a nona hora.
- Dia do mês (de 1 a 31).
- Mês (de 1 a 12).
- Dia da semana (de 0 a 7). Os números 0 e 7 correspondem ao domingo.

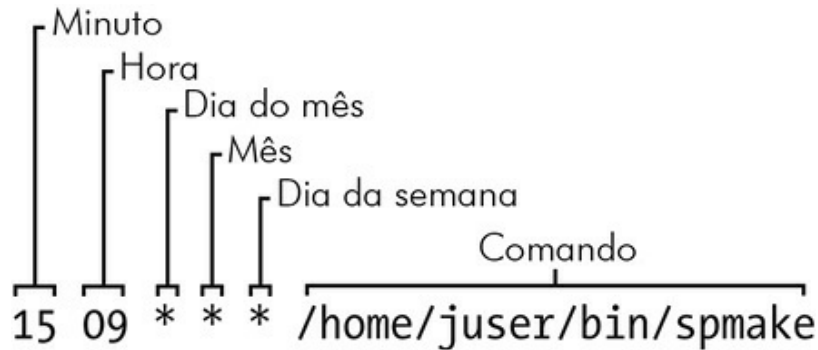


Figura 7.3 – Uma entrada no arquivo crontab.

Um asterisco (*) em qualquer campo significa que pode haver correspondência com qualquer valor. O exemplo anterior executa `spmake` diariamente, pois os campos referentes ao dia do mês, ao mês e ao dia da semana estão todos preenchidos com asterisco, que o cron lê como “execute esse job todos os dias, em todos os meses e em todas as semanas”.

Para executar `spmake` somente no décimo quarto dia de cada mês, utilize a linha de crontab a seguir:

```
15 09 14 * * /home/juser/bin/spmake
```

Você pode selecionar mais de uma data para cada campo. Por exemplo, para executar o programa no quinto e no décimo quarto dia de cada mês, insira 5,14 no terceiro campo:

```
15 09 5,14 * * /home/juser/bin/spmake
```

 **Observação:** se o cron job gerar uma saída-padrão ou um erro, ou sair de forma anormal, o cron deverá enviar essa informação por email a você. Redirecione a saída para `/dev/null` ou para outro arquivo de log se você achar que o email é inconveniente.

A página de manual `crontab(5)` disponibiliza informações completas sobre o formato do crontab.

7.6.1 Instalando arquivos do crontab

Cada usuário pode ter seu próprio arquivo do crontab, o que significa que todo sistema pode ter vários crontabs, normalmente encontrados em `/var/spool/cron/crontabs`. Usuários normais não podem escrever nesse diretório; o comando `crontab` instala, lista, edita e remove o crontab de um usuário.

A maneira mais fácil de instalar um crontab é colocar as entradas de seu crontab em um arquivo e, em seguida, usar `crontab arquivo` para instalar `arquivo` como o seu crontab corrente. O comando `crontab` verifica o formato do arquivo para garantir que você não cometeu nenhum erro. Para listar seus cron jobs, execute `crontab -l`. Para remover o crontab, use `crontab -r`.

No entanto, após ter criado o seu crontab inicial, poderá ser um pouco confuso usar arquivos temporários para efetuar novas edições. Em vez disso, você poderá editar e instalar seu crontab em um único passo usando o comando `crontab -e`. Se você cometer um erro, o `crontab` deverá informar-lhe em que local o erro está e perguntará se você deseja tentar editar o arquivo novamente.

7.6.2 Arquivos crontab do sistema

Em vez de usar o crontab do superusuário para agendar tarefas recorrentes do sistema, as distribuições Linux normalmente têm um arquivo `/etc/crontab`. Não use `crontab` para editar esse arquivo, pois essa versão tem um campo adicional – o usuário que deve executar o job – inserido antes do comando a ser executado. Por exemplo, o cron job a seguir, definido em `/etc/crontab`, executa às 6h42min como superusuário (root, mostrado em ❶).

```
42 6 * * * root❶ /usr/local/bin/cleansystem > /dev/null 2>&1
```

👉 **Observação:** algumas distribuições armazenam arquivos crontab de sistema no diretório `/etc/cron.d`. Esses arquivos podem ter qualquer nome, porém têm o mesmo formato que `/etc/crontab`.

7.6.3 O futuro do cron

O utilitário cron é um dos componentes mais antigos de um sistema Linux; ele tem estado presente há décadas (é anterior ao próprio Linux), e o formato de sua configuração não sofreu muitas alterações durante vários anos. Quando algo se torna tão antigo assim, ele passa a ser candidato a uma substituição, e há esforços em andamento para isso.

As substituições propostas, na realidade, são somente partes de versões mais recentes do init: para o `systemd`, há unidades de timer, e para o Upstart, a ideia é ser capaz de criar eventos recorrentes para disparar jobs. Afinal de contas, ambas as versões de init podem executar tarefas como qualquer usuário, e elas oferecem certas vantagens como logging personalizado.

Entretanto a realidade é que nem o `systemd` nem o Upstart atualmente têm todas as capacidades do cron. Além do mais, quando eles se tornarem capazes, uma compatibilidade com versões anteriores será necessária para suportar tudo que dependa

do cron. Por esses motivos, é pouco provável que o formato do cron deixe de ser utilizado em breve.

7.7 Agendando tarefas a serem executadas uma só vez com at

Para executar um job somente uma vez no futuro sem usar o cron, utilize o serviço at. Por exemplo, para executar myjob às 22h30min, digite o comando a seguir:

```
$ at 22:30
```

```
at> myjob
```

Finalize a entrada com Ctrl-D. (O utilitário at lê os comandos da entrada-padrão.)

Para verificar se o job foi agendado, use atq. Para removê-lo, utilize atrm. Você também pode agendar jobs para que executem em alguns dias no futuro ao adicionar uma data no formato DD.MM.AA, por exemplo, at 22:30 30.09.15.

Não há muito mais a dizer sobre o comando at. Embora at não seja usado com tanta frequência, ele poderá ser útil para aqueles momentos incomuns em que você precise dizer ao sistema para efetuar um desligamento no futuro.

7.8 Entendendo os IDs de usuário e as mudanças de usuário

Já discutimos como programas setuid como sudo e su permitem mudar de usuários, e mencionamos componentes de sistema como o login, que controla o acesso dos usuários. Talvez você esteja se perguntando como essas partes funcionam e que papel tem o kernel na mudança de usuário.

Há duas maneiras de mudar um ID de usuário, e o kernel cuida de ambas. A primeira é com um executável setuid, discutido na seção 2.17. A segunda é por meio da família setuid() de chamadas de sistema. Há algumas versões diferentes dessa chamada de sistema para acomodar os vários IDs de usuário associados a um processo, como você verá na seção 7.8.1.

O kernel tem regras básicas sobre o que um processo pode ou não fazer, porém, a seguir, estão as três regras básicas:

- Um processo executando como root (userid 0) pode usar setuid() para se tornar qualquer outro usuário.
- Um processo que não estiver executando como root tem restrições severas em

relação ao modo como pode usar `setuid()`; na maioria dos casos, ele não poderá fazê-lo.

- Qualquer processo pode executar um programa `setuid`, desde que tenha as permissões de arquivo adequadas.

 **Observação:** a mudança de usuário não tem nada a ver com senhas ou com nomes de usuário. Esses conceitos são estritamente do espaço de usuário, como você viu inicialmente no arquivo `/etc/passwd` na seção 7.3.1. Você conhecerá mais detalhes sobre como isso funciona na seção 7.9.1.

7.8.1 Propriedade de processos, UID efetivo, UID real e UID salvo

Nossa discussão sobre IDs de usuário até agora foi simplificada. Na realidade, todo processo tem mais de um ID de usuário. Descrevemos o *euid* (effective user ID, ou ID de usuário efetivo), que define os direitos de acesso para um processo. Um segundo ID de usuário, o *ruid* (real user ID, ou ID de usuário real), indica quem iniciou um processo. Quando você executa um programa `setuid`, o Linux define o ID de usuário efetivo com o proprietário do programa durante a execução, porém mantém o ID do usuário original no ID de usuário real.

Nos sistemas modernos, a diferença entre os IDs de usuário efetivo e real é confusa, tanto que muita documentação que diz respeito à propriedade de processos está incorreta.

Pense no ID de usuário efetivo como o *ator* e o ID de usuário real como o *proprietário*. O ID de usuário real define o usuário que pode interagir com o processo em execução – mais significativamente, qual usuário pode matar e enviar sinais a um processo. Por exemplo, se o usuário A iniciar um novo processo que execute como usuário B (de acordo com as permissões `setuid`), o usuário A continua sendo o dono do processo e poderá matá-lo.

Em sistemas Linux normais, a maioria dos processos tem o mesmo ID de usuário efetivo e real. Por padrão, `ps` e outros programas de diagnóstico do sistema mostram o ID do usuário efetivo. Para visualizar tanto o ID do usuário efetivo quanto o ID do usuário real em seu sistema, experimente executar o comando a seguir, porém não fique surpreso se descobrir que as duas colunas de IDs de usuário são idênticas para todos os processos de seu sistema:

```
$ ps -eo pid,euser,ruser,comm
```

Para criar uma exceção somente para que você possa ver valores diferentes nas colunas, tente fazer experimentos com a criação de uma cópia `setuid` do comando `sleep`, execute a cópia durante alguns segundos e, em seguida, execute o comando `ps` anterior

em outra janela antes que a cópia termine.

Para colocar mais lenha na fogueira, além dos IDs de usuário real e efetivo, há também um *ID de usuário salvo* (que normalmente não é abreviado). Um processo pode mudar seu ID de usuário efetivo para o ID de usuário real ou salvo durante a execução. [Para complicar mais ainda a situação, o Linux tem ainda outro ID de usuário: o *fsuid* (file system user ID, ou ID de usuário do sistema de arquivos), que define o usuário que estiver acessando o sistema de arquivos, porém esse é raramente usado.]

Comportamento típico de um programa setuid

A ideia do ID de usuário real pode ser contraditória à sua experiência anterior. Por que você não precisa lidar com os outros IDs de usuário com muita frequência? Por exemplo, após iniciar um processo com `sudo`, se quiser matá-lo, você continuará usando `sudo`; você não pode matá-lo como o seu próprio usuário normal. O seu usuário normal não deveria ser o ID do usuário real, nesse caso, fornecendo-lhe as permissões corretas?

O motivo para esse comportamento é que o `sudo` e vários outros programas `setuid` mudam explicitamente os IDs de usuário efetivo e real com uma das chamadas de sistema `setuid()`. Esses programas fazem isso porque, com frequência, há efeitos colaterais indesejados e problemas de acesso quando todos os IDs de usuário não coincidem.

 **Observação:** se você estiver interessado nos detalhes e nas regras relacionadas à mudança de IDs de usuário, leia a página de manual `setuid(2)` e verifique as outras páginas de manual listadas na seção SEE ALSO (Veja também). Há várias chamadas de sistema diferentes para situações diversificadas.

Alguns programas não gostam de ter um ID de usuário real igual a `root`. Para evitar que o `sudo` altere o ID de usuário real, acrescente a linha a seguir em seu arquivo `/etc/sudoers` (e tome cuidado com os efeitos colaterais em outros programas que você queira executar como `root`!):

```
Defaults    stay_setuid
```

Implicações quanto à segurança

Como o kernel do Linux cuida de todas as mudanças de usuário (e como resultado, das permissões de acesso a arquivos) por meio de programas `setuid` e chamadas de sistema subsequentes, os desenvolvedores e administradores de sistema devem ser extremamente cuidadosos em relação a dois aspectos:

- os programas que têm permissões `setuid`
- o que esses programas fazem

Se você criar uma cópia do shell `bash` que seja `setuid root`, qualquer usuário local poderá executá-lo e ter domínio completo do sistema. É realmente simples assim. Além do mais, até mesmo um programa de propósito especial que seja `setuid root` pode representar perigo caso contenha bugs. Explorar pontos fracos de programas que executam como `root` é um método importante para invasão de sistemas, e há exploits demais desse tipo para mencionar.

Pelo fato de haver tantas maneiras de invadir um sistema, evitar isso é uma atividade com várias facetas. Uma das maneiras essenciais de manter atividades indesejadas distante de seu sistema consiste em garantir que a autenticação dos usuários seja feita com nomes de usuário e senhas.

7.9 Identificação e autenticação de usuários

Um sistema multiusuário deve oferecer suporte básico para a segurança dos usuários no que diz respeito à identificação e autenticação. A parte referente à *identificação* da segurança responde à pergunta sobre *quem* são os usuários. A parte sobre *autenticação* pede que os usuários *proven* que são quem eles dizem que são. Por fim, a *autorização* é usada para definir e limitar o que os usuários têm *permissão* para fazer.

Quando se trata de identificação de usuários, o kernel do Linux conhece apenas os IDs numéricos dos usuários para saber quem é dono de processos e de arquivos. O kernel conhece regras de autorização para saber como executar programas `setuid` e como os IDs de usuário podem executar a família `setuid()` de chamadas de sistema para mudar de um usuário para outro. Entretanto o kernel não sabe nada sobre autenticação, ou seja, nomes de usuário, senhas e assim por diante. Praticamente tudo o que está relacionado à autenticação acontece no espaço de usuário.

Discutimos o mapeamento entre IDs de usuário e senhas na seção 7.3.1; agora iremos explicar como os processos de usuário acessam esse mapeamento. Começaremos com um caso excessivamente simplificado, em que um processo de usuário quer saber seu nome de usuário (o nome correspondente ao ID do usuário efetivo). Em um sistema Unix tradicional, um processo pode fazer algo como o que se segue para obter o seu nome de usuário:

1. O processo pede ao kernel o ID do usuário efetivo usando a chamada de sistema `geteuid()`.
2. O processo abre o arquivo `/etc/passwd` e começa a ler do início.
3. O processo lê uma linha do arquivo `/etc/passwd`. Se não sobrar mais nada para ler,

é sinal de que o processo falhou em encontrar o nome do usuário.

4. O processo efetua parse da linha para identificar os campos (separando tudo o que estiver entre dois-pontos). O terceiro campo é o ID do usuário para a linha corrente.
5. O processo compara o ID do passo 4 com o ID do passo 1. Se forem idênticos, o primeiro campo do passo 4 será o nome de usuário desejado; o processo pode parar de pesquisar e poderá usar esse nome.
6. O processo vai para a próxima linha em */etc/passwd* e retorna ao passo 3.

Esse é um procedimento longo que, geralmente, é muito mais complicado na realidade.

7.9.1 Usando bibliotecas para obter informações de usuário

Se todo desenvolvedor que precisasse saber o nome do usuário corrente tivesse que escrever todo o código que acabamos de ver, o sistema seria uma confusão horivelmente desarticulada, cheia de bugs, inchada e impossível de manter. Felizmente, podemos usar bibliotecas-padrão para realizar tarefas repetitivas, portanto tudo o que você terá de fazer normalmente para obter o nome de um usuário é chamar uma função como `getpwuid()` da biblioteca-padrão, depois que você tiver a resposta de `geteuid()`. (Consulte as páginas de manual dessas chamadas para saber mais sobre como elas funcionam.)

Quando a biblioteca-padrão é compartilhada, você pode fazer alterações significativas na implementação, sem alterar nenhum outro programa. Por exemplo, você pode deixar de usar */etc/passwd* para seus usuários e utilizar um serviço de rede como o LDAP em seu lugar.

Essa abordagem tem funcionado bem para identificar nomes de usuário associados a IDs, porém as senhas têm se mostrado mais complicadas. A seção 7.3.1 descreve como, tradicionalmente, a senha criptografada fazia parte de */etc/passwd*, portanto, se você quisesse conferir uma senha fornecida por um usuário, seria necessário criptografar o que quer que o usuário tivesse digitado e comparar esse valor com o conteúdo do arquivo */etc/passwd*.

Essa implementação tradicional tem as seguintes limitações:

- Ela não define um padrão válido para todo o sistema para o protocolo de criptografia.
- Ela supõe que você tem acesso à senha criptografada.
- Ela supõe que você deseja solicitar uma senha ao usuário sempre que esse quiser acessar algo que exija autenticação (o que é irritante).

- Ela supõe que você quer usar senhas. Se você quiser usar tokens com valores únicos, cartões inteligentes, dados biométricos ou outra forma diferente de autenticação de usuário, será necessário acrescentar esse suporte por conta própria.

Algumas dessas limitações contribuíram para o desenvolvimento do pacote de senha shadow discutido na seção 7.3.3, que representou o primeiro passo em direção a uma configuração de senha válida para todo o sistema. Porém a solução para a maior parte dos problemas veio com o design e a implementação do PAM.

7.10 PAM

Para proporcionar flexibilidade à autenticação de usuários, em 1995, a Sun Microsystems propôs um novo padrão chamado *PAM* (Pluggable Authentication Modules) – um sistema de bibliotecas compartilhadas para autenticação (Open Source Software Foundation RFC 86.0, outubro de 1995). Para autenticar um usuário, uma aplicação passa o usuário ao PAM para determinar se esse usuário pode se identificar com sucesso. Dessa maneira, é relativamente fácil acrescentar suporte para técnicas adicionais de autenticação, como a autenticação de dois fatores e chaves físicas. Além do suporte ao sistema de autenticação, o PAM também oferece uma quantidade limitada de controle de autorização para serviços (por exemplo, no caso de você querer proibir um serviço como o cron a determinados usuários).

Pelo fato de haver vários tipos de cenários de autenticação, o PAM emprega diversos *módulos de autenticação* carregáveis dinamicamente. Cada módulo executa uma tarefa específica; por exemplo, o módulo *pam_unix.so* pode conferir a senha de um usuário.

Essa é uma atividade complexa, para dizer o mínimo. A interface de programação não é simples, e não está claro se o PAM realmente resolve todos os problemas existentes. Apesar disso, o suporte ao PAM está em praticamente todos os programas que exijam autenticação em um sistema Linux, e a maioria das distribuições utiliza o PAM. E como ele funciona sobre a API de autenticação existente do Unix, o suporte para integração com um cliente exige pouco trabalho extra, se é que exige algum.

7.10.1 Configuração do PAM

Iremos explorar o básico sobre o funcionamento do PAM ao analisar a sua configuração. Normalmente, você encontrará arquivos de configuração de aplicações PAM no diretório */etc/pam.d* (sistemas mais antigos poderão usar um único arquivo */etc/pam.conf*). A maioria das instalações inclui vários arquivos, portanto você poderá não saber por onde começar. Alguns nomes de arquivo devem corresponder a partes do

sistema que você já conhece, por exemplo, *cron* e *passwd*.

Como a configuração específica nesses arquivos varia significativamente entre as distribuições, pode ser difícil encontrar um exemplo comum. Daremos uma olhada em um exemplo de linha de configuração que você poderá encontrar para *chsh* (o programa *change shell* para mudança de shell):

```
auth    requisite pam_shells.so
```

Essa linha diz que o shell do usuário deve estar em */etc/shells* para que o usuário se autentique com sucesso junto ao programa *chsh*. Vamos ver como isso é feito. Cada linha de configuração tem três campos: um tipo de função, o argumento de controle e o módulo, nessa ordem. Eis o que eles significam nesse exemplo:

- Tipo de função – a função que uma aplicação de usuário pede ao PAM para realizar. Nesse caso, é *auth* – a tarefa de autenticar o usuário.
- Argumento de controle – essa configuração controla o que o PAM faz *após* o sucesso ou a falha de sua ação para a linha corrente (*requisite*, nesse exemplo). Vamos explicar isso em breve.
- Módulo – o módulo de autenticação executado para essa linha, que determina o que a linha realmente faz. Nesse caso, o módulo *pam_shells.so* verifica se o shell corrente do usuário está listado em */etc/shells*.

A configuração do PAM está detalhada na página de manual *pam.conf(5)*. Vamos dar uma olhada em alguns pontos essenciais.

Tipos de função

Uma aplicação de usuário pode pedir ao PAM que realize uma das quatro funções a seguir.

- *auth* – autenticar um usuário (ver se o usuário é quem diz ser).
- *account* – verifica o status da conta do usuário (se o usuário está autorizado a fazer algo, por exemplo).
- *session* – executa algo somente para a sessão corrente do usuário (por exemplo, exibir uma mensagem do dia).
- *password* – muda a senha de um usuário ou outras credenciais.

Para qualquer linha de configuração, o módulo e a função juntos determinam a ação do PAM. Um módulo pode ter mais de um tipo de função, portanto, ao determinar o propósito de uma linha de configuração, lembre-se sempre de considerar a função e o módulo como um par. Por exemplo, o módulo *pam_unix.so* verifica uma senha ao

executar a função `auth`, porém ele define uma senha se a função `for` `password`.

Argumentos de controle e regras empilhadas

Um recurso importante do PAM é que as regras especificadas por suas linhas de configuração podem ser *empilhadas*, o que significa que você pode aplicar muitas regras ao executar uma função. É por esse motivo que o argumento de controle é importante: o sucesso ou a falha de uma ação em uma linha pode causar impactos nas linhas seguintes ou fazer com que a função toda tenha sucesso ou falhe.

Há dois tipos de argumentos de controle: com sintaxe simples e com uma sintaxe mais avançada. A seguir, temos os três principais argumentos de controle com sintaxe simples que poderão ser encontrados em uma regra.

- `sufficient` — se essa regra for bem-sucedida, a autenticação será um sucesso e o PAM não precisará verificar nenhuma outra regra adicional. Se a regra falhar, o PAM prosseguirá com as regras adicionais.
- `requisite` — se essa regra for bem-sucedida, o PAM prosseguirá com as regras adicionais. Se a regra falhar, a autenticação não terá sucesso e o PAM não precisará olhar para outras regras adicionais.
- `required` — se essa regra tiver sucesso, o PAM prosseguirá com as regras adicionais. Se a regra falhar, o PAM prosseguirá com as regras adicionais, porém sempre retornará uma autenticação sem sucesso, independentemente do resultado final das regras adicionais.

Prosseguindo com o exemplo anterior, a seguir, temos uma pilha de exemplo para a função de autenticação de `chsh`:

```
auth    sufficient pam_rootok.so
auth    requisite  pam_shells.so
auth    sufficient pam_unix.so
auth    required   pam_deny.so
```

Com essa configuração, quando o comando `chsh` pedir ao PAM que execute a função de autenticação, o PAM fará o seguinte (veja a figura 7.4, que contém um diagrama de fluxo):

1. O módulo *pam_rootok.so* verifica se o usuário `root` é aquele que está tentando se autenticar. Em caso afirmativo, ele terá sucesso imediato e não tentará mais se autenticar. Isso funciona porque o argumento de controle está definido com `sufficient`, o que significa que o sucesso dessa ação é suficiente para o PAM informar sucesso imediatamente ao `chsh`. Caso contrário, ele prosseguirá para o passo 2.

2. O módulo *pam_shells.so* verifica se o shell do usuário está em */etc/shells*. Se o shell não estiver lá, o módulo retornará uma falha e o argumento de controle *requisite* indica que o PAM deve informar essa falha imediatamente de volta a *chsh*, e não deve tentar novas autenticações. Caso contrário, o shell está em */etc/shells*, portanto o módulo retornará sucesso e atenderá à flag de controle de *requisite*; prossiga para o passo 3.
3. O módulo *pam_unix.so* solicita a senha ao usuário e a confere. O argumento de controle está definido como *sufficient*, portanto o sucesso desse módulo (uma senha correta) é suficiente para o PAM informar sucesso ao *chsh*. Se a senha estiver incorreta, o PAM continuará em direção ao passo 4.
4. O módulo *pam_deny.so* sempre falha e, como o argumento de controle *required* está presente, o PAM informa ao *chsh* que houve falha. Esse é o default quando não houver mais nada a ser tentado. (Observe que um argumento de controle igual a *required* não faz com que o PAM deixe a função falhar imediatamente – ele executará qualquer linha que sobrar em sua pilha –, mas o que será informado à aplicação será sempre uma falha.)

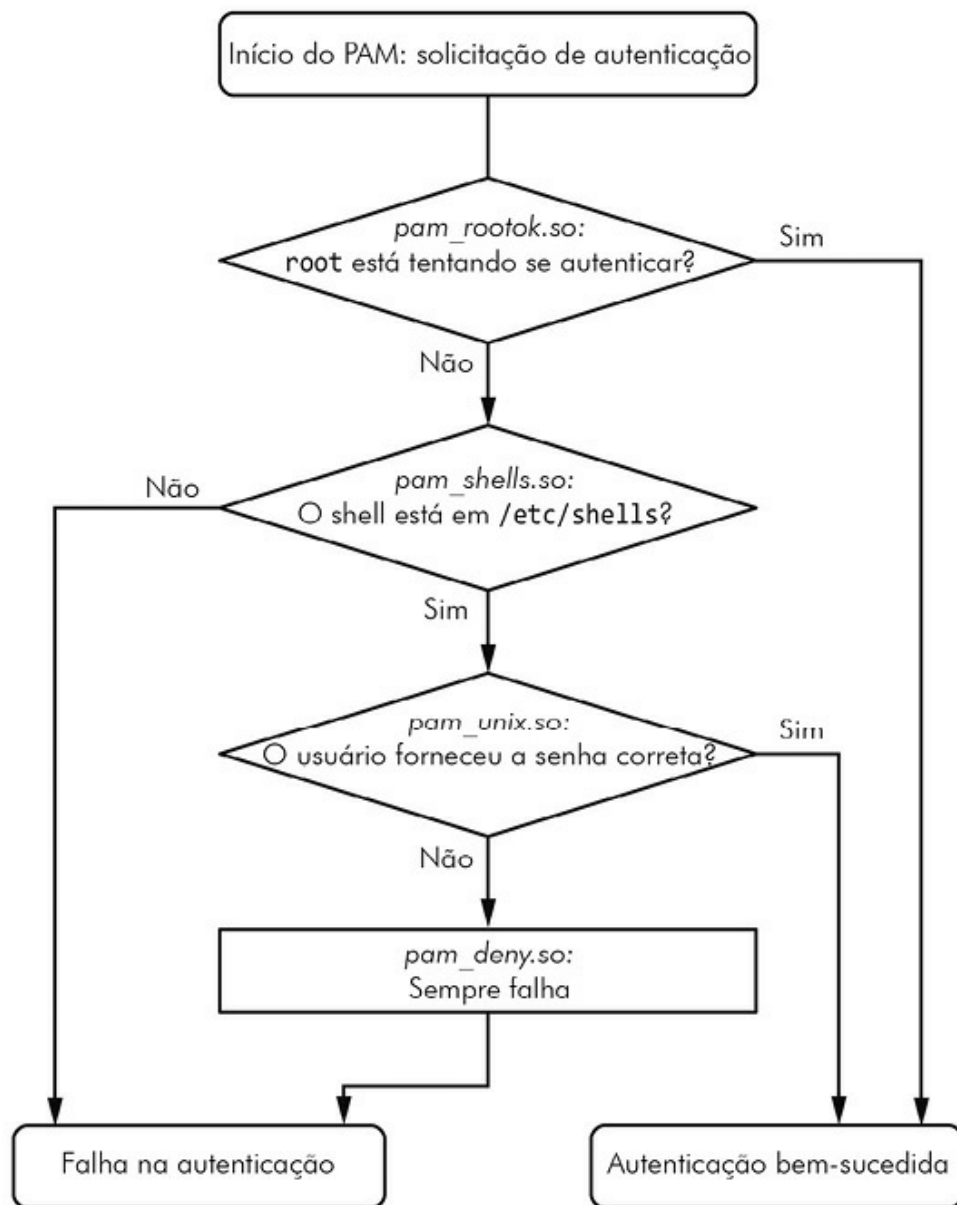


Figura 7.4 – Fluxo de execução das regras do PAM.

👍 **Observação:** não confunda os termos *função* e *ação* ao trabalhar com o PAM. A função é o objetivo de alto nível: o que a aplicação do usuário quer que o PAM faça (autenticar um usuário, por exemplo). Uma ação é um passo específico que o PAM executa para alcançar esse objetivo. Basta se lembrar de que a aplicação do usuário chama primeiro a função e que o PAM cuida das particularidades com as ações.

A sintaxe avançada do argumento de controle, representada entre colchetes (`[]`), permite controlar manualmente uma reação, de acordo com o valor específico retornado pelo módulo (e não apenas sucesso ou fracasso). Para obter os detalhes, consulte a página de manual `pam.conf(5)`; quando tiver compreendido a sintaxe simples, você não terá problemas com a sintaxe avançada.

Argumentos de módulos

Os módulos do PAM podem aceitar argumentos após o nome do módulo. Com

frequência, você encontrará o exemplo a seguir com o módulo *pam_unix.so*:

```
auth    sufficient pam_unix.so nullok
```

O argumento *nullok*, nesse caso, diz que o usuário pode não ter senha (o default seria falhar se o usuário não tiver nenhuma senha).

7.10.2 Observações sobre o PAM

Devido à sua capacidade de fluxo de controle e à sintaxe de argumentos de módulo, a sintaxe de configuração do PAM tem várias funcionalidades de uma linguagem de programação e certo grau de eficácia. Mal tocamos a superfície até agora, mas a seguir estão algumas dicas relacionadas ao PAM:

- Para descobrir quais módulos do PAM estão presentes em seu sistema, experimente usar `man -k pam_` (observe o underscore). Pode ser difícil descobrir a localização dos módulos. Experimente usar o comando `locate unix_pam.so` e veja o resultado que você consegue obter.
- As páginas de manual contêm as funções e os argumentos para cada módulo.
- Muitas distribuições geram automaticamente determinados arquivos de configuração do PAM, portanto alterá-los diretamente em */etc/pam.d* pode não ser uma atitude sábia. Leia os comentários nos arquivos em */etc/pam.d* antes de editá-los; se forem arquivos gerados, os comentários lhe dirão de onde eles vieram.
- O arquivo de configuração */etc/pam.d/other* define a configuração default para qualquer aplicação que não tenha o seu próprio arquivo de configuração. O default geralmente é proibir tudo.
- Há diferentes maneiras de incluir arquivos de configuração adicionais em um arquivo de configuração do PAM. A sintaxe `@include` carrega um arquivo de configuração completo, porém um argumento de controle para carregar somente a configuração para uma função em particular também pode ser utilizado. O uso varia entre as distribuições.
- A configuração do PAM não termina com os argumentos de módulo. Alguns módulos podem acessar arquivos adicionais em */etc/security*, geralmente para configurar restrições por usuário.

7.10.3 PAM e senhas

Em virtude da evolução da verificação de senhas no Linux ao longo dos anos, vários artefatos para configuração de senha que às vezes causam confusão continuam existindo. O primeiro com o qual é preciso tomar cuidado é o arquivo */etc/login.defs*.

Esse é o arquivo de configuração para o pacote de senha shadow original. Ele contém informações sobre o algoritmo de criptografia usado no arquivo de senha shadow, porém é raramente usado em um sistema moderno com PAM instalado, pois a configuração do PAM contém essa informação. Apesar do que foi dito, o algoritmo de criptografia em */etc/login.defs* deve coincidir com a configuração do PAM nos raros casos em que você se deparar com uma aplicação que não suporte o PAM.

Em que lugar o PAM obtém suas informações sobre o esquema de criptografia de senhas? Lembre-se de que há duas maneiras de o PAM interagir com senhas: a função `auth` (para conferir uma senha) e a função `password` (para definir uma senha). É mais fácil localizar o parâmetro para configuração de senha. A melhor maneira provavelmente é executar um `grep`:

```
$ grep password.*unix /etc/pam.d/*
```

As linhas correspondentes devem conter *pam_unix.so* e terão um aspecto semelhante a:

```
password    sufficient    pam_unix.so obscure sha512
```

Os argumentos `obscure` e `sha512` dizem ao PAM o que fazer quando uma senha for definida. Em primeiro lugar, o PAM verifica se a senha é “obscura” o suficiente (ou seja, se a senha não é muito parecida com a senha antiga, entre outras verificações); em seguida, o PAM utiliza o algoritmo SHA512 para criptografar a nova senha.

Porém isso acontece *somente* quando um usuário *define* uma senha, e não quando o PAM está *verificando* uma. Então como o PAM sabe qual algoritmo deve usar ao efetuar uma autenticação? Infelizmente, a configuração não lhe dirá nada; não há nenhum argumento de criptografia para *pam_unix.so* para a função `auth`. As páginas de manual também não dizem nada.

O fato é que (na época desta publicação) *pam_unix.so* simplesmente tentava adivinhar o algoritmo, normalmente pedindo à biblioteca `libcrypt` que fizesse o trabalho sujo de tentar várias opções até que algo funcionasse ou não houvesse mais nada a ser tentado. Sendo assim, geralmente, não será preciso se preocupar com o algoritmo de criptografia para verificação.

7.11 Próximos passos

Estamos aproximadamente na metade de nossa jornada por este livro, e já discutimos muitos dos blocos de construção vitais de um sistema Linux. A discussão sobre logging e usuários em um sistema Linux apresentou aquilo que torna possível dividir os serviços e as tarefas em porções pequenas e independentes que ainda sabem como interagir até certo ponto.

Este capítulo lidou quase exclusivamente com o espaço de usuário, e agora precisamos refinar nossa visão sobre os processos do espaço de usuário e os recursos que eles consomem. Para isso, retornaremos ao kernel no capítulo 8.

CAPÍTULO 8

Observando mais de perto os processos e a utilização de recursos

Este capítulo detalha mais os relacionamentos entre processos, o kernel e os recursos do sistema. Há três tipos básicos de recursos de hardware: CPU, memória e I/O. Os processos competem por esses recursos e o trabalho do kernel é alocá-los de forma justa. O próprio kernel também é um recurso – um recurso de software que os processos usam para realizar tarefas como criar novos processos e se comunicar com outros.

Com frequência, muitas das ferramentas que você verá neste capítulo são consideradas como ferramentas de monitoração de desempenho. Elas são particularmente úteis se o seu sistema estiver se tornando muito lento e você estiver tentando descobrir por quê. No entanto não se deixe distrair demais pelo desempenho; em geral, tentar otimizar um sistema que já esteja funcionando corretamente é perda de tempo. Em vez disso, concentre-se em entender o que as ferramentas realmente avaliam, e você terá uma boa ideia de como o kernel funciona.

8.1 Monitorando processos

Você aprendeu a usar o `ps` na seção 2.16 para listar os processos que estão executando em seu sistema em um determinado instante. O comando `ps` lista os processos correntes, porém não faz muito para informar como os processos mudam ao longo do tempo. Desse modo, ele não irá realmente ajudar você a determinar quais processos estão utilizando muito tempo de CPU ou memória.

O programa `top` geralmente é mais útil que o `ps`, pois exibe o status atual do sistema bem como muitos dos campos de uma listagem de `ps`, além de atualizar a saída a cada segundo. Talvez o aspecto mais importante seja o fato de `top` mostrar os processos mais ativos (ou seja, aqueles que estão consumindo a maior parte do tempo de CPU no momento) no início de sua saída.

Você pode enviar comandos para `top` usando teclas. A seguir estão alguns dos comandos mais importantes:

- Barra de espaço – atualiza imediatamente a saída.
- M – ordena de acordo com o uso atual de memória residente.
- T – ordena de acordo com o total (cumulativo) de uso de CPU.
- P – ordena de acordo com o uso corrente de CPU (default).
- u – exibe somente os processos de um usuário.
- f – seleciona estatísticas diferentes para exibir.
- ? – mostra um resumo do uso de todos os comandos de `top`.

Dois outros utilitários do Linux semelhantes ao `top` oferecem um conjunto mais sofisticado de visualizações e de recursos: `atop` e `htop`. A maioria dos recursos extras está disponível em outros utilitários. Por exemplo, o `htop` tem várias funcionalidades do comando `lsof`, que será descrito na próxima seção.

8.2 Encontrando arquivos abertos com `lsof`

O comando `lsof` lista arquivos abertos e os processos que os estão utilizando. Como o Unix dá bastante ênfase aos arquivos, `lsof` está entre as ferramentas mais úteis para identificar pontos com problemas. Porém o `lsof` não serve somente para arquivos normais – ele pode listar recursos de rede, bibliotecas dinâmicas, pipes etc.

8.2.1 Lendo a saída de `lsof`

Executar `lsof` na linha de comando normalmente gera uma quantidade enorme de dados de saída. A seguir, apresentamos um fragmento do que você poderá ver. Essa saída inclui arquivos abertos pelo processo `init` bem como por um processo `vi` em execução:

```
$ lsof
COMMAND PID USER  FD  TYPE DEVICE  SIZE  NODE NAME
init    1 root  cwd  DIR   8,1  4096    2 /
init    1 root  rtd  DIR   8,1  4096    2 /
init    1 root  mem  REG   8, 47040 9705817 /lib/i386-linux-gnu/libnss_files-2.15.so
init    1 root  mem  REG   8,1 42652 9705821 /lib/i386-linux-gnu/libnss_nis-2.15.so
init    1 root  mem  REG   8,1 92016 9705833 /lib/i386-linux-gnu/libnsl-2.15.so
--trecho omitido--
vi 22728 juser  cwd  DIR   8,1 4096 14945078 /home/juser/w/c
vi 22728 juser 4u  REG   8,1 1288 1056519 /home/juser/w/c/f
--trecho omitido--
```

A saída mostra os campos a seguir (listados na linha superior):

- COMMAND – o nome do comando para o processo que detém o descritor de

arquivo.

- PID – o ID do processo.
- USER – o usuário executando o processo.
- FD – esse campo pode conter dois tipos de elemento. Na saída anterior, a coluna FD mostra o propósito do arquivo. Esse campo também pode listar o *descritor de arquivo* do arquivo aberto – um número usado por um processo, em conjunto com as bibliotecas de sistema e o kernel, para identificar e manipular um arquivo.
- TYPE – o tipo do arquivo (arquivo normal, diretório, socket e assim por diante).
- DEVICE – os números principal (major) e secundário (minor) do dispositivo que detém o arquivo.
- SIZE – o tamanho do arquivo.
- NODE – o número do inode do arquivo.
- NAME – o nome do arquivo.

A página de manual `lsuf(1)` contém uma lista completa do que você poderá ver em cada campo, mas você deverá ser capaz de descobrir o que é que está sendo visto somente observando a saída. Por exemplo, observe as entradas com `cwd` no campo FD, conforme destacadas em negrito. Essas linhas indicam os diretórios de trabalho corrente dos processos. Outro exemplo é a última linha, que mostra um arquivo que o usuário está editando no momento com `vi`.

8.2.2 Usando o `lsuf`

Há duas abordagens básicas para executar o `lsuf`:

- Liste tudo e faça pipe da saída para um comando como `less` e, em seguida, pesquise o que você estiver procurando. Isso pode demorar um pouco por causa do volume de dados de saída gerados.
- Restrinja a lista fornecida por `lsuf` usando opções de linha de comando.

As opções de linha de comando podem ser usadas para fornecer um nome de arquivo como argumento e fazer o `lsuf` listar somente as entradas que correspondam ao argumento. Por exemplo, o comando a seguir exibe entradas para arquivos abertos em `/usr`:

```
$ lsuf /usr
```

Para listar os arquivos abertos associados a um ID de processo em particular, execute:

```
$ lsuf -p pid
```

Para obter um pequeno resumo das várias opções de `lsf`, execute `lsf -h`. A maioria das opções diz respeito ao formato da saída. (Consulte o capítulo 10 para ver uma discussão sobre os recursos de rede do `lsf`.)

👍 **Observação:** `lsf` é altamente dependente de informações do kernel. Se você fizer uma atualização de seu kernel e não estiver atualizando tudo rotineiramente, talvez seja necessário fazer um upgrade do `lsf`. Além do mais, se você fizer uma atualização de distribuição tanto do kernel quanto do `lsf`, o `lsf` atualizado poderá não funcionar até que o sistema seja reiniciado com o novo kernel.

8.3 Execução de programas de tracing e chamadas de sistema

As ferramentas que vimos até agora analisam processos ativos. Contudo, se você não tiver ideia do motivo pelo qual um programa morre quase imediatamente após ter iniciado, nem mesmo o `lsf` poderá ajudá-lo. Com efeito, você terá dificuldades mesmo se executar `lsf` de forma concorrente com um comando que falhar.

Os comandos `strace` (trace de chamadas de sistema) e `ltrace` (trace de biblioteca) podem ajudar você a descobrir o que um programa está tentando fazer. Essas ferramentas geram quantidades extraordinárias de dados de saída, porém, uma vez que você saiba o que está procurando, haverá mais ferramentas à sua disposição para rastrear os problemas.

8.3.1 `strace`

Lembre-se de que uma *chamada de sistema* é uma operação privilegiada que um processo do espaço de usuário pede ao kernel para executar, por exemplo, abrir e ler dados de um arquivo. O utilitário `strace` exibe todas as chamadas de sistema feitas por um processo. Para vê-lo em ação, execute o comando a seguir:

```
$ strace cat /dev/null
```

No capítulo 1, você aprendeu que quando um processo quer iniciar outro, ele faz a chamada de sistema `fork()` para gerar uma cópia de si mesmo e, em seguida, a cópia usa um membro da família de chamadas de sistema `exec()` para começar a execução de um novo programa. O comando `strace` começa trabalhando no novo processo (a cópia do processo original), imediatamente após a chamada a `fork()`. Desse modo, as primeiras linhas da saída desse comando devem mostrar `execve()` em ação, seguido de uma chamada de inicialização de memória `brk()`, como se segue:

```
execve("/bin/cat", ["cat", "/dev/null"], [/* 58 vars */]) = 0
brk(0)                                = 0x9b65000
```


A próxima parte da saída está relacionada principalmente com a carga de bibliotecas compartilhadas. Esses dados podem ser ignorados, a menos que você realmente queira saber o que o sistema de bibliotecas compartilhadas está fazendo.

```
access("/etc/ld.so.nohwcap", F_OK)    = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb77b5000
access("/etc/ld.so.preload", R_OK)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
--trecho omitido--
open("/lib/libc.so.6", O_RDONLY)      = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\200^\1"... , 1024) = 1024
```

Além disso, pule os dados de saída referentes à mmap até chegar às linhas que tenham um aspecto semelhante a:

```
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 6), ...}) = 0
open("/dev/null", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFCHR|0666, st_rdev=makedev(1, 3), ...}) = 0
fadvise64_64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
read(3, "", 32768)          = 0
close(3)                    = 0
close(1)                    = 0
close(2)                    = 0
exit_group(0)               = ?
```

Essa parte da saída mostra o comando em funcionamento. Inicialmente, dê uma olhada na chamada a `open()`, que abre um arquivo. 3 é um resultado que significa sucesso (3 é o descritor do arquivo retornado pelo kernel após o arquivo ter sido aberto). Abaixo disso, você pode notar o momento em que `cat` lê de `/dev/null` (a chamada a `read()`, que também tem 3 como descritor do arquivo). Em seguida, não há mais nada para ler, portanto o programa fecha o descritor de arquivo e sai com `exit_group()`.

O que acontece quando ocorre um problema? Experimente usar `strace cat not_a_file` no lugar do comando anterior e verifique a chamada a `open()` na saída resultante:

```
open("not_a_file", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or directory)
```

Como `open()` não pôde abrir o arquivo, ele retornou -1 para sinalizar um erro. Você pode ver que `strace` informa o erro exato e fornece uma pequena descrição dele.

Arquivos faltando representam os problemas mais comuns em programas Unix, portanto, se o log do sistema e as informações de outros logs não forem muito úteis e você não tiver mais para onde recorrer, `strace` poderá ser de grande ajuda. Ele pode até mesmo ser usado em daemons que se desassociam. Por exemplo:

```
$ strace -o crummyd_strace -ff crummyd
```

Nesse exemplo, a opção `-o` de `strace` faz com que seja feito log da ação de qualquer processo-filho gerado por `crummysd` em `crummysd_strace.pid`, em que *pid* é o ID de processo do processo-filho.

8.3.2 ltrace

O comando `ltrace` monitora chamadas a bibliotecas compartilhadas. A saída é semelhante àquela de `strace`, motivo pelo qual estamos mencionando-o aqui, porém esse comando não monitora nada no nível do kernel. Considere-se avisado de que há *muito mais* chamadas a bibliotecas compartilhadas do que chamadas de sistema. Definitivamente, você deverá filtrar a saída, e o próprio `ltrace` tem várias opções prontas para dar assistência a você.

👉 **Observação:** consulte a seção 15.1.4 para obter mais informações sobre bibliotecas compartilhadas. O comando `ltrace` não funciona em binários cujo link tenha sido feito estaticamente.

8.4 Threads

No Linux, alguns processos são divididos em partes chamadas *threads*. Uma thread é muito semelhante a um processo – ela tem um identificador (TID, ou thread ID), e o kernel escalona e executa threads do mesmo modo como faz com os processos. No entanto, de modo diferente de processos separados, que normalmente não compartilham recursos do sistema como memória e conexões de I/O com outros processos, todas as threads em um único processo compartilham recursos de sistema e memória.

8.4.1 Processos single-threaded e multithreaded

Muitos processos têm somente uma thread. Um processo com uma thread é *single-threaded*, e um processo com mais de uma thread é *multithreaded*. Todos os processos começam como single-threaded. Essa thread inicial geralmente é chamada de *thread principal* (main thread). A thread principal pode então iniciar novas threads para que o processo se torne multithreaded, de maneira semelhante ao modo como um processo pode chamar `fork()` para iniciar um novo processo.

👉 **Observação:** é raro fazer qualquer referência a threads quando um processo for single-threaded. Este livro não mencionará threads, a menos que processos multithreaded façam diferença no que você vir ou experimentar.

A principal vantagem de um processo multithreaded é que, quando o processo tiver muito trabalho a fazer, as threads poderão executar simultaneamente em vários processadores, potencialmente agilizando o processamento. Embora você também possa conseguir fazer processamentos simultâneos com vários processos, as threads são

mais rápidas para iniciar quando comparadas aos processos e, com frequência, é mais fácil e/ou mais eficiente para as threads se comunicarem entre si usando sua memória compartilhada do que os processos se comunicarem por meio de um canal como uma conexão de rede ou um pipe.

Alguns programas usam threads para resolver problemas de administração de vários recursos de I/O. Tradicionalmente, um processo às vezes poderia usar `fork()` para iniciar um novo subprocesso a fim de lidar com um novo stream de entrada ou de saída. As threads oferecem um sistema semelhante, sem o overhead de iniciar um novo processo.

8.4.2 Visualizando as threads

Por padrão, a saída dos comandos `ps` e `top` mostra somente os processos. Para exibir informações de threads com `ps`, adicione a opção `m`. Aqui está um exemplo de saída:

```
$ ps m
  PID TTY          STAT TIME COMMAND
 3587 pts/3    -      0:00 bash❶
    - -      Ss      0:00 -
 3592 pts/4    -      0:00 bash❷
    - -      Ss      0:00 -
12287 pts/8    -      0:54 /usr/bin/python /usr/bin/gm-notify❸
    - -      SLl      0:48 -
    - -      SLl      0:00 -
    - -      SLl      0:06 -
    - -      SLl      0:00 -
```

Listagem 8.1 – Visualizando as threads com `ps m`.

A listagem 8.1 mostra os processos, juntamente com as threads. Cada linha com um número na coluna PID (em ❶, ❷ e ❸) representa um processo, como na saída normal de um `ps`. As linhas com traços na coluna PID representam as threads associadas ao processo. Nessa saída, os processos em ❶ e em ❷ têm apenas uma thread cada, porém o processo 12287 em ❸ é multithreaded, com quatro threads.

Se quiser ver os IDs das threads usando `ps`, um formato personalizado de saída poderá ser usado. O exemplo a seguir mostra somente os IDs dos processos, os IDs das threads e o comando:

Listagem 8.2 – Mostrando os IDs dos processos e os IDs das threads com `ps m`

```
$ ps m -o pid,tid,command
  PID  TID  COMMAND
 3587   -   bash
```

```
- 3587 -  
3592 - bash  
- 3592 -  
12287 - /usr/bin/python /usr/bin/gm-notify  
- 12287 -  
- 12288 -  
- 12289 -  
- 12295 -
```

O exemplo de saída na listagem 8.2 corresponde às threads mostradas na listagem 8.1. Observe que os IDs das threads de processos single-threaded são idênticos aos IDs dos processos; essa é a thread principal. Para o processo multithreaded 12287, a thread 12287 também é a thread principal.

 **Observação:** normalmente, você não irá interagir com threads individuais como faria com os processos. Você precisaria saber muito sobre o modo como um programa multithreaded foi escrito para poder atuar em uma thread de cada vez, e, mesmo assim, fazer isso pode não ser uma boa ideia.

As threads podem provocar confusão no que diz respeito à monitoração de recursos, pois as threads individuais em um processo multithreaded podem consumir recursos simultaneamente. Por exemplo, o `top`, por padrão, não mostra as threads; será preciso teclar `H` para ativar esse recurso. Na maioria das ferramentas de monitoração de recursos que você está prestes a conhecer, será preciso um pouco de trabalho extra para habilitar a exibição de threads.

8.5 Introdução à monitoração de recursos

Agora iremos discutir alguns tópicos relacionados à monitoração de recursos, incluindo tempo de processador (CPU), memória e I/O de disco. Analisaremos a utilização no nível de todo o sistema bem como por processo.

Muitas pessoas se envolvem com o funcionamento interno do kernel do Linux visando à melhoria de desempenho. Entretanto a maioria dos sistemas Linux funciona bem com as configurações default de uma distribuição, e você pode passar dias tentando ajustar o desempenho de seu computador sem obter resultados significativos, especialmente se não souber o que deve procurar. Portanto, em vez de pensar no desempenho como visto nas ferramentas deste capítulo, pense em ver o kernel em ação enquanto ele divide os recursos entre os processos.

8.6 Medindo o tempo de CPU

Para monitorar um ou mais processos específicos ao longo do tempo, utilize a opção `-p`

de `top`, com a sintaxe a seguir:

```
$ top -p pid1 [-p pid2 ...]
```

Para descobrir quanto tempo de CPU um comando usa durante o seu tempo de vida, utilize `time`. A maioria dos shells tem um comando `time` incluído que não fornece estatísticas extensas, portanto é provável que você vá precisar executar `/usr/bin/time`. Por exemplo, para medir o tempo de CPU usado por `ls`, execute:

```
$ /usr/bin/time ls
```

Depois que `ls` terminar, `time` deverá exibir uma saída como a que está sendo mostrada a seguir. Os campos principais estão em **negrito**:

```
0.05user 0.09system 0:00.44elapsed 31%CPU (0avgtext+0avgdata 0maxresident)k  
0inputs+0outputs (125major+51minor)pagefaults 0swaps
```

- **User time** (tempo de usuário): a quantidade de segundos que a CPU gastou executando o *próprio* código do programa. Nos processadores modernos, alguns comandos executam tão rapidamente – e desse modo, o tempo de CPU é tão baixo – que `time` arredonda o valor para zero.
- **System time** (tempo de sistema): a quantidade de tempo que o kernel gasta realizando o trabalho do processo (por exemplo, lendo arquivos e diretórios).
- **Elapsed time** (tempo decorrido): o tempo total gasto para executar o processo, do início ao fim, incluindo o tempo que a CPU gastou realizando outras tarefas. Esse número normalmente não é muito útil para medir o desempenho, porém subtrair o tempo de usuário e o tempo de sistema do tempo decorrido pode proporcionar uma ideia geral de quanto tempo um processo fica esperando por recursos do sistema.

O restante da saída detalha principalmente o uso de memória e de I/O. Você aprenderá mais sobre a saída referente a page fault na seção 8.9.

8.7 Ajustando as prioridades do processo

Podemos mudar o modo como o kernel escalona um processo para dar mais ou menos tempo de CPU a um processo em relação a outros. O kernel executa cada processo de acordo com sua *prioridade* de escalonamento, que é um número entre -20 e 20, com -20 sendo a prioridade mais alta. (Sim, isso pode ser confuso.)

O comando `ps -l` lista a prioridade atual de um processo, porém é um pouco mais fácil ver as prioridades em ação com o comando `top`, como mostrado a seguir:

```
$ top
```

```
Tasks: 244 total,  2 running, 242 sleeping,  0 stopped,  0 zombie
```

Cpu(s): 31.7%us, 2.8%sy, 0.0%ni, 65.4%id, 0.2%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 6137216k total, 5583560k used, 553656k free, 72008k buffers
Swap: 4135932k total, 694192k used, 3441740k free, 767640k cached

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
28883	bri	20	0	1280m	763m	32m	S	58	12.7	213:00.65	chromium-browser
1175	root	20	0	210m	43m	28m	R	44	0.7	14292:35	Xorg
4022	bri	20	0	413m	201m	28m	S	29	3.4	3640:13	chromium-browser
4029	bri	20	0	378m	206m	19m	S	2	3.5	32:50.86	chromium-browser
3971	bri	20	0	881m	359m	32m	S	2	6.0	563:06.88	chromium-browser
5378	bri	20	0	152m	10m	7064	S	1	0.2	24:30.21	compiz
3821	bri	20	0	312m	37m	14m	S	0	0.6	29:25.57	soffice.bin
4117	bri	20	0	321m	105m	18m	S	0	1.8	34:55.01	chromium-browser
4138	bri	20	0	331m	99m	21m	S	0	1.7	121:44.19	chromium-browser
4274	bri	20	0	232m	60m	13m	S	0	1.0	37:33.78	chromium-browser
4267	bri	20	0	1102m	844m	11m	S	0	14.1	29:59.27	chromium-browser
2327	bri	20	0	301m	43m	16m	S	0	0.7	109:55.65	unity-2d-shell

Na saída anterior de `top`, a coluna `PR` (prioridade) lista a prioridade atual de escalonamento do kernel para o processo. Quanto maior o número, menos provável será que o kernel vá escalonar o processo se outros precisarem de tempo de CPU. A prioridade de escalonamento sozinha não determina a decisão do kernel de conceder tempo de CPU a um processo, e ela muda frequentemente durante a execução dos programas de acordo com a quantidade de tempo de CPU que o processo consumir.

Ao lado da coluna de prioridade está a coluna *nice value* (`NI`), que dá uma pista ao escalonador do kernel. É com isso que você deve se preocupar ao tentar influenciar a decisão do kernel. O kernel soma o *nice value* à prioridade corrente para determinar o próximo time slot (fatia de tempo) para o processo.

Por padrão, *nice value* é igual a 0. Mas suponha que você esteja fazendo bastante processamento em background e não queira que sua sessão interativa fique mais lenta. Para fazer aquele processo dar lugar a outros e executar somente quando as outras tarefas não tiverem nada para fazer, você pode alterar o valor de *nice value* para 20 por meio do comando `renice` (em que *pid* é o ID do processo que você quer alterar):

```
$ renice 20 pid
```

Se você for o superusuário, poderá definir *nice value* com um valor negativo, porém fazer isso quase sempre não é uma boa ideia, pois os processos do sistema poderão não obter tempo de CPU suficiente. Com efeito, provavelmente não será necessário alterar muito o *nice value* porque muitos sistemas Linux têm somente um único usuário, e esse usuário não realiza muito processamento de verdade. (O *nice value* era bem mais importante no passado, quando havia vários usuários em um único computador.)

8.8 Médias de carga

O desempenho de CPU é uma das métricas mais simples de medir. A *média de carga* (load average) é o número médio de processos prontos para executar no momento. Ou seja, é uma estimativa do número de processos que podem usar a CPU em um instante qualquer. Ao pensar na média de carga, tenha em mente que a maioria dos processos em seu sistema normalmente estará esperando uma entrada (do teclado, do mouse ou da rede, por exemplo), o que significa que a maioria dos processos não estará pronta para executar e não deverá contribuir com nada para a média de carga. Somente os processos que realmente estão fazendo algo afetarão a média de carga.

8.8.1 Utilizando o uptime

O comando `uptime` informa três médias de carga, além do tempo que o kernel está executando:

```
$ uptime
```

```
... up 91 days, ... load average: 0.08, 0.03, 0.01
```

Os três números em negrito correspondem às médias de carga do último minuto, dos 5 últimos minutos e dos 15 últimos minutos, respectivamente. Como você pode ver, esse sistema não está muito ocupado: uma média de apenas 0,01 processo executou em todos os processadores nos últimos 15 minutos. Em outras palavras, se você tivesse apenas um processador, ele estaria executando aplicações do espaço de usuário durante 1% dos últimos 15 minutos. (Tradicionalmente, a maioria dos sistemas desktop exibiria uma média de carga de aproximadamente 0 se você estivesse realizando qualquer atividade, *exceto* compilar um programa ou utilizar um jogo. Uma média de carga igual a 0 normalmente é um bom sinal, pois significa que o seu processador não está sendo desafiado e que você está economizando energia.)

👉 **Observação:** os componentes da interface de usuário em sistemas desktop atuais tendem a ocupar mais CPU do que no passado. Por exemplo, em sistemas Linux, um plugin Flash de um navegador web pode ser particularmente um notório consumidor de recursos, e aplicações Flash podem facilmente ocupar boa parte da CPU e da memória de um sistema em decorrência de uma implementação pobre.

Se uma média de carga subir e ficar em torno de 1, um único processo provavelmente estará usando a CPU quase todo o tempo. Para identificar esse processo, utilize o comando `top`; o processo normalmente será apresentado no início da saída.

A maioria dos sistemas modernos tem mais de um processador principal ou CPU, de modo que vários processos podem facilmente executar de forma simultânea. Se você tiver dois cores, uma média de carga igual a 1 significa que somente um dos cores provavelmente estará ativo em um instante qualquer, e uma média de carga igual a 2

significa que ambos os cores têm o suficiente para fazer o tempo todo.

8.8.2 Cargas altas

Uma média de carga alta não significa necessariamente que seu sistema está tendo problemas. Um sistema com recursos de memória e de I/O suficientes pode facilmente lidar com vários processos em execução. Se sua média de carga está alta, mas seu sistema continua respondendo bem, não entre em pânico: o sistema simplesmente tem muitos processos compartilhando a CPU. Os processos devem competir uns com os outros pelo tempo de processador e, como resultado, eles demoram mais para realizar seus processamentos do que levariam se cada um deles pudesse usar a CPU o tempo todo. Outro caso em que você poderá ver uma média de carga alta como normal é em um servidor web, em que os processos podem iniciar e terminar tão rapidamente que o sistema de avaliação da média de carga não conseguirá funcionar de modo eficiente.

Entretanto, se você perceber que o sistema está lento e que a média de carga está alta, é sinal de que pode haver problemas de desempenho com a memória. Quando o sistema estiver com pouca memória, o kernel poderá iniciar um processo de *thrashing*, ou seja, fazer swap de memória dos processos rapidamente, de e para o disco. Quando isso ocorrer, muitos processos ficarão prontos para executar, porém suas memórias poderão não estar disponíveis, de modo que eles permanecerão em um estado pronto para executar (e contribuirão com a média de carga) durante muito mais tempo do que normalmente ficariam.

Daremos uma olhada agora na memória de forma muito mais detalhada.

8.9 Memória

Uma das maneiras mais simples de verificar o status de memória de seu sistema como um todo é executar o comando `free` ou visualizar `/proc/meminfo` e ver a quantidade de memória real que está sendo usada para caches e buffers. Como acabamos de mencionar, problemas de desempenho podem surgir por falta de memória. Se não houver muita memória para cache/buffer sendo usada (e o restante da memória real estiver ocupada), você poderá precisar de mais memória. Contudo é fácil demais culpar a falta de memória por todos os problemas de desempenho de seu computador.

8.9.1 Como a memória funciona

Lembre-se de que, de acordo com o capítulo 1, a CPU tem uma MMU (Memory Management Unit, ou Unidade de gerenciamento de memória) que traduz os endereços

de memória virtual usados pelos processos em endereços reais. O kernel dá assistência ao MMU dividindo a memória usada pelos processos em partes menores chamadas *páginas*. O kernel mantém uma estrutura de dados chamada *tabela de páginas* (page table) que contém um mapeamento dos endereços de páginas virtuais de um processo para endereços de páginas reais na memória. À medida que um processo acessa a memória, a MMU traduz os endereços virtuais usados por esse processo para endereços reais, de acordo com a tabela de páginas do kernel.

Um processo de usuário não precisa que todas as suas páginas estejam imediatamente disponíveis para poder executar. O kernel geralmente carrega e aloca as páginas à medida que um processo precisar delas; esse sistema é conhecido como *paginação por demanda* (on-demand paging ou simplesmente demand paging). Para ver como isso funciona, considere o modo como um programa inicia e executa como um novo processo:

1. O kernel carrega o início do código com as instruções do programa em páginas da memória.
2. O kernel pode alocar algumas páginas de memória de trabalho para o novo processo.
3. À medida que executar, o processo poderá alcançar um ponto em que a próxima instrução de seu código não estará em nenhuma das páginas carregadas inicialmente pelo kernel. Nesse ponto, o kernel assume o controle, carrega as páginas necessárias para a memória e, em seguida, permite que o programa retome a execução.
4. De modo semelhante, se o programa exigir mais memória de trabalho do que foi inicialmente alocada, o kernel cuidará disso ao encontrar páginas livres (ou criando espaço para elas) e atribuindo-as ao processo.

8.9.2 Page faults

Se uma página de memória não estiver pronta quando um processo quiser usá-la, o processo irá disparar um *page fault* (falha de página). No caso de um page fault, o kernel assumirá o controle da CPU para preparar a página. Há dois tipos de page faults: minor (secundário) e major (principal).

Page faults minor

Um page fault minor ocorre quando a página desejada está na memória principal, porém a MMU não sabe em que local ela está. Isso pode ocorrer quando o processo solicita

mais memória ou quando a MMU não tem espaço suficiente para armazenar todas as localizações de páginas para um processo. Nesse caso, o kernel informa à MMU sobre a página e permite que o processo continue. Os page faults minors não são um problema sério, e muitos ocorrem à medida que um processo executa. A menos que você precise de um desempenho máximo para algum programa que faça uso intensivo de memória, é provável que você não vá precisar se preocupar com eles.

Page faults major

Um page fault major ocorre quando a página de memória desejada não está na memória principal, o que significa que o kernel deve carregá-la do disco ou de outro sistema de armazenamento lento. Muitos page faults major deixarão o sistema lento, pois o kernel deverá realizar uma quantidade substancial de trabalho para disponibilizar as páginas, tirando a chance de os processos normais executarem.

Alguns page faults major não podem ser evitados, por exemplo, aqueles que ocorrem quando o código é carregado do disco ao executar um programa pela primeira vez. Os problemas mais sérios ocorrem quando você começa a ficar sem memória e o kernel passa a fazer *swap* das páginas da memória de trabalho para o disco a fim de criar espaço para novas páginas.

Observando os page faults

Você pode explorar os page faults de processos individuais usando os comandos `ps`, `top` e `time`. O comando a seguir mostra um exemplo simples de como o comando `time` exibe os page faults. (A saída do comando `cal` não é importante, portanto estamos descartando-a ao redirecioná-la para `/dev/null`.)

```
$ /usr/bin/time cal > /dev/null
0.00user 0.00system 0:00.06elapsed 0%CPU (0avgtext+0avgdata 3328maxresident)k
648inputs+0outputs (2major+254minor)pagefaults 0swaps
```

Como você pode ver pelo texto em negrito, quando esse programa executou, houve 2 page faults major e 254 minor. Os page faults major ocorreram quando o kernel precisou carregar o programa do disco pela primeira vez. Se você executasse o comando novamente, é provável que não haveria nenhum page fault major, pois o kernel teria carregado as páginas do disco para cache.

Se preferir ver os page faults dos processos à medida que estiverem executando, utilize `top` ou `ps`. Ao executar `top`, utilize `f` para mudar os campos exibidos e `u` para exibir a quantidade de page faults major. (O resultado será mostrado em uma nova coluna `nFLT`. Você não verá os page faults minor.)

Ao utilizar `ps`, um formato personalizado de saída poderá ser usado para ver os page faults de um processo em particular. A seguir, apresentamos um exemplo para o ID de processo 20365:

```
$ ps -o pid,minflt,majflt 20365
PID MINFL MAJFL
20365 834182 23
```

As colunas MINFL e MAJFL mostram as quantidades de page faults minor e major. É claro que você pode combinar isso com quaisquer outras opções de seleção de processos, conforme descrito na página de manual `ps(1)`.

Visualizar page faults por processo pode ajudar você a identificar determinados componentes problemáticos. Entretanto, se estiver interessado no desempenho de seu sistema como um todo, você precisará de uma ferramenta para sintetizar as ações de CPU e de memória para todos os processos.

8.10 Monitorando o desempenho da CPU e da memória com `vmstat`

Entre as diversas ferramentas disponíveis para monitorar o desempenho do sistema, o comando `vmstat` é um dos mais antigos, com um mínimo de overhead. Você o achará útil para obter uma visão geral da frequência com que o kernel está efetuando swap de páginas, o nível de ocupação da CPU e a utilização de I/O.

O truque para usar toda a eficácia do `vmstat` está em entender a sua saída. Por exemplo, a seguir, apresentamos uma saída de `vmstat 2`, que informa estatísticas a cada dois segundos:

```
$ vmstat 2
procs -----memory----- ---swap-- -----io---- -system-- -----cpu----
r b swpd free buff cache si so bi bo in cs us sy id wa
2 0 320416 3027696 198636 1072568 0 0 1 1 2 0 15 2 83 0
2 0 320416 3027288 198636 1072564 0 0 0 1182 407 636 1 0 99 0
1 0 320416 3026792 198640 1072572 0 0 0 58 281 537 1 0 99 0
0 0 320416 3024932 198648 1074924 0 0 0 308 318 541 0 0 99 1
0 0 320416 3024932 198648 1074968 0 0 0 0 208 416 0 0 99 0
0 0 320416 3026800 198648 1072616 0 0 0 0 207 389 0 0 100 0
```

A saída se divide em categorias: `procs` para processos, `memory` para uso de memória, `swap` para as páginas que entram e saem da área de swap, `io` para uso de disco, `system` para o número de vezes que o kernel alterna para o código do kernel e `cpu` para o tempo usado pelas diferentes partes do sistema.

A saída anterior é típica de um sistema que não está executando muitas tarefas. Normalmente, você começará observando a segunda linha da saída – a primeira é uma média do tempo em que o sistema está ativo. Por exemplo, nesse caso, o sistema tem 320.416 KB de memória que sofreu swap para o disco (*swpd*) e aproximadamente 3.025.000 KB (3 GB) de memória real livre (*free*). Embora um pouco de área de swap esteja em uso, as colunas *si* (swap-in) e *so* (swap-out) com valores iguais a zero informam que o kernel não está fazendo swap de nada no momento, de e para o disco. A coluna *buff* indica a quantidade de memória que o kernel está usando para buffers de disco (veja a seção 4.2.5).

Na extremidade direita, embaixo do cabeçalho CPU, você pode ver a distribuição do tempo de CPU nas colunas *us*, *sy*, *id* e *wa*. Essas colunas listam (em ordem) o percentual de tempo que a CPU está gastando em tarefas de usuário, em tarefas de sistema (kernel), o tempo de idle e o tempo de espera de I/O. No exemplo anterior, não há muitos processos de usuário executando (eles estão usando um máximo de 1% de CPU); o kernel não está fazendo praticamente nada, enquanto a CPU está tranquila, não fazendo nada durante 99% do tempo.

Agora observe o que acontece quando um programa grande é iniciado algum tempo depois (as duas primeiras linhas ocorrem imediatamente antes de o programa executar):

Listagem 8.3 – Atividade da memória

```
procs -----memory----- ---swap-- -----io---- -system-- ----cpu----
r b swpd free buff cache si so bi bo in cs us sy id wa
1 0 320412 2861252 198920 1106804 0 0 0 0 2477 4481 25 2 72 0❶
1 0 320412 2861748 198924 1105624 0 0 0 40 2206 3966 26 2 72 0
1 0 320412 2860508 199320 1106504 0 0 210 18 2201 3904 26 2 71 1
1 1 320412 2817860 199332 1146052 0 0 19912 0 2446 4223 26 3 63 8
2 2 320284 2791608 200612 1157752 202 0 4960 854 3371 5714 27 3 51 18❷
1 1 320252 2772076 201076 1166656 10 0 2142 1190 4188 7537 30 3 53 14
0 3 320244 2727632 202104 1175420 20 0 1890 216 4631 8706 36 4 46 14
```

Como você pode ver em ❶ na listagem 8.3, a CPU passa a ter um pouco de uso durante um longo período, especialmente por causa dos processos de usuário. Como há memória livre suficiente, a quantidade de espaço de cache e de buffer usada começa a aumentar à medida que o kernel passa a usar mais o disco.

Mais tarde, vemos algo interessante: observe em ❷ que o kernel traz algumas páginas para a memória que haviam sido removidas pelo swap (a coluna *si*). Isso significa que o programa que acabou de executar provavelmente acessou algumas páginas compartilhadas por outro processo. Isso é comum; muitos processos usam o código de

determinadas bibliotecas compartilhadas somente quando estão inicializando.

Observe também que, de acordo com a coluna *b*, alguns processos estão *bloqueados* (impedidos de executar) enquanto esperam por páginas de memória. Em geral, a quantidade de memória livre está diminuindo, porém não está nem próxima de se esgotar. Há também uma boa dose de atividade em disco, conforme visto pelos números crescentes nas colunas *bi* (blocks in) e *bo* (blocks out).

A saída é bem diferente quando a memória se esgota. À medida que o espaço livre acaba, os tamanhos tanto do buffer quanto de cache diminuem, pois o kernel precisa cada vez mais do espaço para os processos de usuário. Quando não restar mais nada, você começará a ver atividades na coluna *so* (swapped out), à medida que o kernel passa a transferir páginas para o disco, momento em que quase todas as demais colunas da saída mudarão para refletir o volume de trabalho que o kernel estará fazendo. Você verá mais tempo de sistema, mais dados de e para o disco e mais processos bloqueados porque a memória que eles querem usar não estará disponível (sofreu swap para disco).

Não explicamos todas as colunas da saída do comando `vmstat`. Você pode explorá-las melhor na página de manual `vmstat(8)`, porém deverá aprender mais sobre o gerenciamento de memória do kernel antes, em uma aula ou em um livro como *Operating System Concepts*, 9ª edição (Wiley, 2012), para entendê-las.

8.11 Monitoração de I/O

Por padrão, `vmstat` mostra algumas estatísticas gerais de I/O. Embora seja possível obter dados bem detalhados sobre o uso de recursos por partição usando `vmstat -d`, você obterá muitos dados de saída com essa opção, o que poderá ser desanimador. Em vez disso, tente começar com uma ferramenta chamada `iostat`, que é somente para I/O.

8.11.1 Usando o `iostat`

Assim como o `vmstat`, quando executado sem nenhuma opção, o `iostat` mostra as estatísticas para o tempo em que seu computador está ativo:

```
$ iostat
```

```
[informações do kernel]
```

```
avg-cpu:  %user  %nice %system %iowait  %steal   %idle
           4.46   0.01   0.67   0.31   0.00  94.55
```

```
Device:            tps    kB_read/s    kB_wrtn/s    kB_read  kB_wrtn
sda                 4.67         7.28        49.86    9493727    65011716
sde                 0.00         0.00         0.00        1230         0
```

A parte referente a `avg-cpu` no início contém as mesmas informações de utilização de

CPU apresentadas por outros utilitários que vimos neste capítulo, portanto pule para a parte inferior, que mostra o seguinte para cada dispositivo:

- tps – número médio de transferências de dados por segundo.
- kB_read/s – número médio de kilobytes lidos por segundo.
- kB_wrtn/s – número médio de kilobytes escritos por segundo.
- kB_read – número total de kilobytes lidos.
- kB_wrtn – número total de kilobytes escritos.

Outra semelhança com `vmstat` está no fato de que você poder fornecer um argumento referente a um intervalo, por exemplo `iostat 2`, para fazer uma atualização a cada dois segundos. Ao usar um intervalo, você poderá exibir somente o relatório de dispositivos se usar a opção `-d` (por exemplo, `iostat -d 2`).

Por padrão, a saída de `iostat` omite informações de partição. Para mostrar todas as informações de partição, utilize a opção `-p ALL`. Como há muitas partições em um sistema normal, você obterá muitos dados na saída. A seguir, apresentamos uma parte do que você poderá ver:

\$ iostat -p ALL

--trecho omitido--

Device:	tps	kB_read/s	kB_wrtn/s	kB_read	kB_wrtn
---------	-----	-----------	-----------	---------	---------

--trecho omitido--

sda	4.67	7.27	49.83	9496139	65051472
sda1	4.38	7.16	49.51	9352969	64635440
sda2	0.00	0.00	0.00	6	0
sda5	0.01	0.11	0.32	141884	416032
sdd0	0.00	0.00	0.00	0	0

--trecho omitido--

sde	0.00	0.00	0.00	1230	0
-----	------	------	------	------	---

Nesse exemplo, `sda1`, `sda2` e `sda5` são partições do disco `sda`, portanto haverá um pouco de sobreposição entre as colunas de leitura e de escrita. Entretanto a soma das colunas das partições não será necessariamente igual à coluna do disco. Embora uma leitura de `sda1` também conte como uma leitura de `sda`, tenha em mente que você poderá ler diretamente de `sda`, por exemplo, quando ler a tabela de partição.

8.11.2 Utilização de I/O e monitoração por processo: `iotop`

Se você precisar ir mais a fundo ainda para ver os recursos de I/O usados por processos individuais, a ferramenta `iotop` poderá ajudar. Usar o `iotop` é semelhante a usar o `top`. Há uma atualização contínua da saída, que mostra os processos que usam mais I/O,

com um resumo geral no início:

```
# iotop
```

```
Total DISK READ: 4.76 K/s | Total DISK WRITE: 333.31 K/s
```

TID	PRI	USER	DISK READ	DISK WRITE	SWAPIN	IO>	COMMAND
260	be/3	root	0.00 B/s	38.09 K/s	0.00 %	6.98 %	[jbd2/sda1-8]
2611	be/4	juser	4.76 K/s	10.32 K/s	0.00 %	0.21 %	zeitgeist-daemon
2636	be/4	juser	0.00 B/s	84.12 K/s	0.00 %	0.20 %	zeitgeist-fts
1329	be/4	juser	0.00 B/s	65.87 K/s	0.00 %	0.03 %	soffice.b~ash-pipe=6
6845	be/4	juser	0.00 B/s	812.63 B/s	0.00 %	0.00 %	chromium-browser
19069	be/4	juser	0.00 B/s	812.63 B/s	0.00 %	0.00 %	rhythmbox

Juntamente com as colunas de usuário, de comando e de leitura/escrita, observe que há uma coluna TID (thread ID), no lugar de um ID de processo. A ferramenta `iotop` é um dos poucos utilitários que exibe threads em vez de processos.

A coluna PRIO (prioridade) indica a prioridade do I/O. É semelhante à prioridade da CPU que já vimos, porém ela afeta a velocidade com que o kernel escalona leituras e escritas de I/O para o processo. Em uma prioridade como `be/4`, a parte referente a `be` é a *classe de escalonamento*, e o número é o nível da prioridade. Como ocorre com as prioridades de CPU, números menores são mais importantes; por exemplo, o kernel concede mais tempo de I/O a um processo com `be/3` do que a um processo com `be/4`.

O kernel utiliza a classe de escalonamento para acrescentar mais controle ao escalonamento de I/O. Você verá três classes de escalonamento em `iotop`:

- `be` – best-effort (melhor esforço). O kernel faz o melhor que puder para escalonar o I/O de forma justa para essa classe. A maioria dos processos executa com essa classe de escalonamento de I/O.
- `rt` – real-time (tempo real). O kernel escalona qualquer I/O de tempo real antes de qualquer outra classe de I/O, não importa o que aconteça.
- `idle` – idle (inativo). O kernel realiza I/O para essa classe somente quando não houver nenhum outro I/O a ser feito. Não há nenhum nível de prioridade para a classe de escalonamento `idle`.

Você pode verificar e alterar a prioridade de I/O de um processo usando o utilitário `ionice`; consulte a página de manual `ionice(1)` para obter detalhes. Contudo é provável que você jamais tenha de se preocupar com a prioridade de I/O.

8.12 Monitoração por processo com `pidstat`

Vimos como podemos monitorar processos específicos com utilitários como `top` e `iotop`.

No entanto essa saída é atualizada com o passar do tempo, e cada atualização apaga a saída anterior. O utilitário `pidstat` permite ver o consumo de recursos de um processo ao longo do tempo no estilo do `vmstat`. A seguir, apresentamos um exemplo simples de monitoração do processo 1329, atualizado a cada segundo:

```
$ pidstat -p 1329 1
```

```
Linux 3.2.0-44-generic-pae (duplex) 07/01/2015 _i686_ (4 CPU)
```

	PID	%usr	%system	%guest	%CPU	CPU	Command
09:26:55 PM							
09:27:03 PM	1329	8.00	0.00	0.00	8.00	1	myprocess
09:27:04 PM	1329	0.00	0.00	0.00	0.00	3	myprocess
09:27:05 PM	1329	3.00	0.00	0.00	3.00	1	myprocess
09:27:06 PM	1329	8.00	0.00	0.00	8.00	3	myprocess
09:27:07 PM	1329	2.00	0.00	0.00	2.00	3	myprocess
09:27:08 PM	1329	6.00	0.00	0.00	6.00	2	myprocess

A saída default mostra os percentuais de tempo de usuário e de sistema, além do percentual geral de tempo de CPU; a saída mostra até mesmo em que CPU o processo estava executando. (A coluna `%guest` é, de certo modo, curiosa – é o percentual de tempo que o processo gastou executando algo em uma máquina virtual. A menos que você esteja executando uma máquina virtual, não se preocupe com isso.)

Embora `pidstat` mostre a utilização de CPU por padrão, ele pode fazer muito mais. Por exemplo, a opção `-r` pode ser usada para monitorar a memória, e `-d`, para habilitar a monitoração de disco. Experimente usar essas opções e, em seguida, dê uma olhada na página de manual `pidstat(1)` para ver mais opções para threads, mudança de contexto ou simplesmente sobre qualquer outro assunto que discutimos neste capítulo.

8.13 Tópicos adicionais

Um dos motivos pelos quais há tantas ferramentas para medir a utilização de recursos é que uma grande variedade de tipos de recurso é consumida de vários modos diferentes. Neste capítulo, vimos CPU, memória e I/O como recursos de sistema sendo consumidos por processos, threads dentro de processos e pelo kernel.

O outro motivo pelo qual as ferramentas existem é que os recursos são *limitados* e, para um sistema ter um bom desempenho, seus componentes devem se esforçar para consumir menos recursos. No passado, muitos usuários compartilhavam um computador, de modo que era necessário garantir que cada usuário tivesse uma quota justa de recursos. Atualmente, embora um computador desktop moderno possa não ter vários usuários, ele continua tendo muitos processos que competem pelos recursos. Da mesma maneira, servidores de rede de alto desempenho exigem uma monitoração

intensa dos recursos do sistema.

Tópicos adicionais relacionados à monitoração de recursos e à análise de desempenho incluem:

- `sar` (System Activity Reporter) – o pacote `sar` contém vários dos recursos de monitoração contínua do `vmstat`, porém registra também a utilização de recursos ao longo do tempo. Com o `sar`, você pode observar um instante em particular no passado para ver o que o seu sistema estava fazendo. Isso é prático quando houver um evento do sistema no passado que você queira analisar.
- `acct` (Process accounting) – o pacote `acct` pode registrar os processos e sua utilização de recursos.
- Quotas – você pode limitar vários recursos do sistema por processo ou por usuário. Consulte `/etc/security/limits.conf` para ver algumas das opções para CPU e memória; há também uma página de manual `limits.conf(5)`. Esse é um recurso do PAM, portanto os processos estão sujeitos a ele somente se foram iniciados com algo que utilize o PAM (por exemplo, um shell de login). Também é possível limitar a quantidade de espaço em disco que um usuário pode usar por meio do sistema `quota`.

Se você estiver interessado em ajustes e desempenho de sistemas em particular, o livro *Systems Performance: Enterprise and the Cloud* de Brendan Gregg (Prentice Hall, 2013) oferece muito mais detalhes.

Também não discutimos ainda muitas e muitas ferramentas que podem ser usadas para monitorar a utilização de recursos de rede. Para usá-las, inicialmente você deverá entender o funcionamento de uma rede. É esse assunto que discutiremos a seguir.

CAPÍTULO 9

Entendendo a rede e sua configuração

A ligação em rede consiste da prática de conectar computadores e enviar dados entre eles. Soa como algo simples, porém, para entender como a rede funciona, você deve fazer duas perguntas fundamentais:

- Como o computador que está enviando os dados sabe para *onde* deve enviá-los?
- Quando o computador destino recebe os dados, como ele sabe *o que* ele acabou de receber?

Um computador responde a essas perguntas usando uma série de componentes, cada qual responsável por um determinado aspecto do envio, da recepção e da identificação dos dados. Os componentes estão organizados em grupos que compõem *camadas de rede* (network layers), que se empilham umas sobre as outras para formar um sistema completo. O kernel do Linux trata a rede de modo semelhante à forma como trata o subsistema SCSI, descrito no capítulo 3.

Como cada camada tende a ser independente, é possível criar redes com várias combinações diferentes dos componentes. É nesse ponto que a configuração da rede pode se tornar bem complicada. Por esse motivo, começaremos este capítulo dando uma olhada nas camadas em redes bem simples. Você aprenderá a ver as configurações de sua própria rede e, quando entender o funcionamento básico de cada camada, estará pronto para aprender a configurar essas camadas por conta própria. Por fim, você prosseguirá para tópicos mais avançados como criar suas próprias redes e configurar firewalls. (Pule esse material se os seus olhos começarem a ficar embaçados; é sempre possível retornar depois.)

9.1 Básico sobre redes

Antes de entrar na teoria sobre camadas de rede, dê uma olhada na rede simples mostrada na figura 9.1.

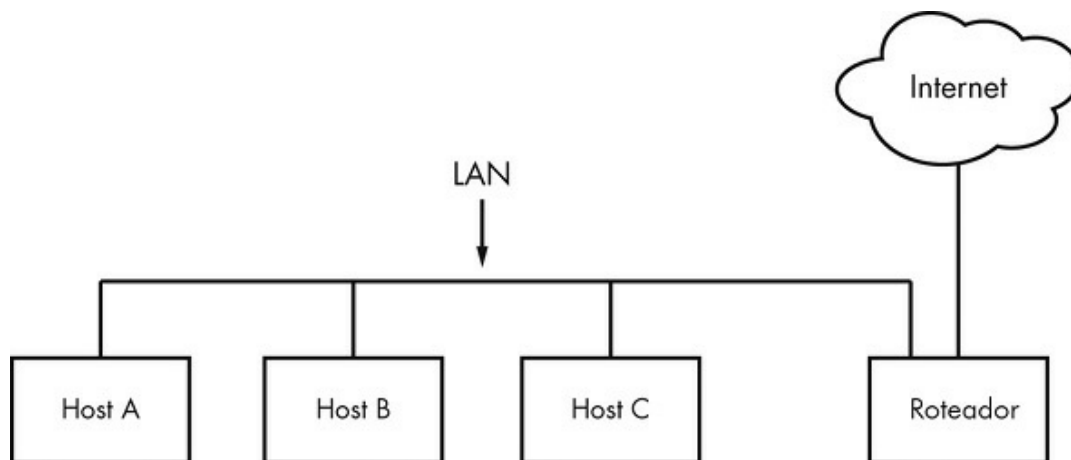


Figura 9.1 – Uma rede local típica, com um roteador que provê acesso à Internet.

Esse tipo de rede está em toda parte; a maioria das redes domésticas e de pequenos escritórios é configurada dessa maneira. Cada computador conectado à rede é chamado de *host*. Os hosts são conectados a um *roteador*, que é um host que pode transferir dados de uma rede para outra. Esses computadores (nesse caso, os Hosts A, B e C) e o roteador formam uma LAN (Local Area Network, ou Rede de área local). As conexões na LAN podem ser com ou sem fio.

O roteador também está conectado à Internet – a nuvem na figura. Como o roteador está conectado tanto à LAN quanto à Internet, todos os computadores da LAN também têm acesso à Internet por meio do roteador. Um dos objetivos deste capítulo é ver como o roteador provê esse acesso.

Seu ponto de vista inicial será a partir de um computador baseado em Linux, como o Host A da LAN na figura 9.1.

9.1.1 Pacotes

Um computador transmite dados por uma rede em pequenas porções chamadas *pacotes*, que são constituídas de duas partes: um *cabeçalho* (header) e um *payload*. O cabeçalho contém informações de identificação, como os hosts de origem/destino e o protocolo básico. O payload, por outro lado, corresponde aos dados propriamente ditos da aplicação, que o computador quer enviar (por exemplo, HTML ou dados de imagem).

Os pacotes permitem que um host se comunique com outros “simultaneamente”, pois os hosts podem enviar, receber e processar pacotes em qualquer ordem, independentemente de onde eles vierem ou para onde estão indo. Dividir mensagens em unidades menores também facilita detectar e efetuar compensações decorrentes de erros na transmissão.

Na maior parte das vezes, você não precisará se preocupar em efetuar a tradução entre

os pacotes e os dados usados pela sua aplicação, pois o sistema operacional tem recursos que fazem isso por você. Entretanto é útil saber qual é a função dos pacotes nas camadas de rede que você está prestes a conhecer.

9.2 Camadas de rede

Uma rede totalmente funcional inclui um conjunto completo de camadas de rede que é chamado de *pilha da rede* (network stack). Qualquer rede funcional tem uma pilha. A pilha típica da Internet, da camada superior para a inferior, tem o seguinte aspecto:

- Camada de aplicação (Application layer) – contém a “linguagem” que as aplicações e os servidores usam para se comunicar: normalmente, é algum tipo de protocolo de alto nível. Protocolos comuns da camada de aplicação incluem o HTTP (Hypertext Transfer Protocol, ou Protocolo de transferência de hipertexto, usado na Web), o SSL (Secure Socket Layer, ou Camada segura de sockets) e o FTP (File Transfer Protocol, ou Protocolo de transferência de arquivos). Os protocolos da camada de aplicação geralmente podem ser combinados. Por exemplo, o SSL é comumente usado em conjunto com o HTTP.
- Camada de transporte (Transport layer) – define as características da transmissão de dados da camada de aplicação. Essa camada inclui verificação da integridade dos dados, portas de origem e de destino e as especificações para dividir os dados de aplicação em pacotes (se a camada de aplicação ainda não tiver feito isso). O TCP (Transmission Control Protocol, ou Protocolo de controle de transmissão) e o UDP (User Datagram Protocol) são os protocolos mais comuns da camada de transporte. Às vezes, essa camada também é chamada de *camada de protocolo* (protocol layer).
- Camada de rede ou de Internet (Network layer ou Internet layer) – define como transferir os pacotes de um host de origem para um host de destino. A regra particular de transmissão de pacotes definida para a Internet é conhecida como IP (Internet Protocol, ou Protocolo de Internet). Como discutiremos somente as redes Internet neste livro, falaremos somente sobre a camada de Internet. Entretanto, como as camadas de rede foram concebidas para serem independentes do hardware, podemos configurar várias camadas de rede independentes de forma simultânea (por exemplo, IP, IPv6, IPX e AppleTalk) em um único host.
- Camada física (Physical layer) – define como enviar dados brutos por um meio físico, por exemplo, Ethernet ou um modem. Às vezes, essa camada é chamada de *camada de enlace* (link layer) ou *camada de host para rede* (host-to-network

layer).

É importante entender a estrutura de uma pilha de rede porque seus dados deverão trafegar por essas camadas pelo menos duas vezes antes de alcançar um programa no destino. Por exemplo, se você estiver enviando dados do Host A para o Host B, conforme mostrado na figura 9.1, seus bytes deixarão a camada de aplicação do Host A e trafegarão pelas camadas de transporte e de rede no Host A; em seguida, eles descerão para o meio físico, passarão por ele e subirão novamente, passando pelos vários níveis mais baixos até a camada de aplicação no Host B, de modo muito semelhante. Se você estiver enviando dados a um host pela Internet por meio do roteador, esses dados passarão por algumas das camadas no roteador (mas, normalmente, não por todas), e por tudo o mais que estiver no caminho.

Às vezes, as camadas se sobrepõem umas às outras de maneira inusitada porque pode ser ineficiente processar todas elas na sequência. Por exemplo, os dispositivos que historicamente lidavam somente com a camada física, hoje em dia, às vezes olham os dados da camada de transporte e de Internet para filtrar e encaminhar os dados rapidamente. (Não se preocupe com isso quando estiver aprendendo o básico.)

Começaremos dando uma olhada em como o seu computador Linux se conecta à rede para responder à pergunta relacionada a *onde* no início do capítulo. Essa é a parte mais baixa da pilha – as camadas física e de rede. Posteriormente, daremos uma olhada nas duas camadas superiores que respondem à pergunta relacionada a *o quê*.

👉 **Observação:** você já deve ter ouvido falar de outro conjunto de camadas conhecido como OSI (Open Systems Interconnection) Reference Model, ou Modelo de referência para interconexão de sistemas abertos. É um modelo de rede de sete camadas, usado frequentemente para ensinar sobre redes e efetuar seu design, porém não discutiremos o modelo OSI porque você trabalhará diretamente com as quatro camadas descritas aqui. Para aprender *muito* mais sobre camadas (e sobre redes em geral), consulte o livro *Redes de computadores* de Andrew S. Tanenbaum e David J. Wetherall, 5ª edição (Pearson, 2011).

9.3 A camada de Internet

Em vez de começarmos com a camada física na parte mais baixa da pilha de rede, iniciaremos com a camada de rede porque ela pode ser mais fácil de entender. A Internet, como a conhecemos atualmente, é baseada no Internet Protocol (Protocolo de Internet) versão 4 (IPv4), embora a versão 6 (IPv6) esteja ganhando adeptos. Um dos aspectos mais importantes da camada de Internet é que ela foi criada para ser uma rede de software que não impõe nenhum requisito em particular sobre o hardware ou os sistemas operacionais. A ideia é que você possa enviar e receber pacotes de Internet sobre qualquer tipo de hardware usando qualquer sistema operacional.

A topologia da Internet é descentralizada; ela é composta de redes menores chamadas *sub-redes*. A ideia é que todas as sub-redes estejam interconectadas de alguma maneira. Por exemplo, na figura 9.1, a LAN normalmente é uma única sub-rede.

Um host pode estar conectado a mais de uma sub-rede. Como vimos na seção 9.1, esse tipo de host é chamado de roteador se ele puder transmitir dados de uma sub-rede para outra (outro termo para roteador é *gateway*). A figura 9.2 detalha a figura 9.1 ao identificar a LAN como uma sub-rede, bem como os endereços de Internet de cada host e do roteador. O roteador da figura tem dois endereços: 10.23.2.1 na sub-rede local e o link para a Internet (porém esse endereço de link para a Internet não é importante nesse momento, portanto está simplesmente identificado como “Endereço de uplink”). Inicialmente, daremos uma olhada nos endereços e, em seguida, na notação de sub-rede.

Cada host de Internet tem pelo menos um *endereço IP* numérico no formato *a.b.c.d*, por exemplo, 10.23.2.37. Um endereço nessa notação é chamado de sequência *dotted-quad*. Se um host estiver conectado a várias sub-redes, ele terá pelo menos um endereço IP por sub-rede. O endereço IP de cada host deve ser único em toda a Internet, porém, como você verá mais adiante, as redes privadas e o NAT podem tornar isso um pouco confuso.

👍 **Observação:** tecnicamente, um endereço IP é constituído de quatro bytes (ou 32 bits): *abcd*. Os bytes *a* e *d* são números de 1 a 254, enquanto *b* e *c* são números de 0 a 255. Um computador processa endereços IP como bytes puros. Entretanto é muito mais fácil para um ser humano ler e escrever um endereço dotted-quad como 10.23.2.37, em vez de algo desagradável como o hexadecimal 0x0A170225.

Endereços IP são como endereços de correspondência, em alguns aspectos. Para se comunicar com outro host, seu computador deve saber o endereço IP desse outro host.

Vamos dar uma olhada no endereço de seu computador.

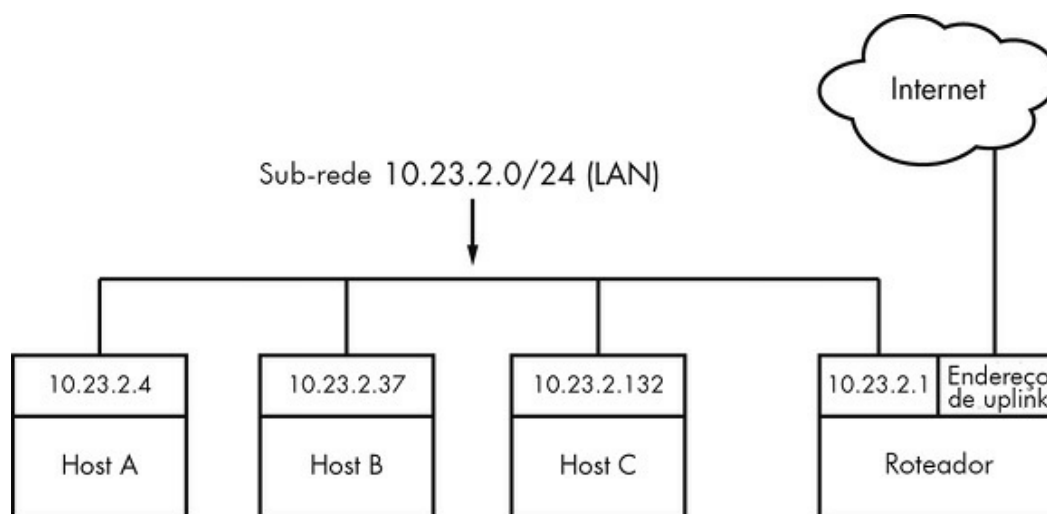


Figura 9.2 – Rede com endereços IP.

9.3.1 Visualizando os endereços IP de seu computador

Um host pode ter vários endereços IP. Para ver os endereços que estão ativos em seu computador Linux, execute:

```
$ ifconfig
```

Provavelmente, haverá muitos dados de saída, porém esses devem incluir algo como:

```
eth0    Link encap:Ethernet HWaddr 10:78:d2:eb:76:97
        inet addr:10.23.2.4 Bcast:10.23.2.255 Mask:255.255.255.0
        inet6 addr: fe80::1278:d2ff:feeb:7697/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
        RX packets:85076006 errors:0 dropped:0 overruns:0 frame:0
        TX packets:68347795 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:86427623613 (86.4 GB) TX bytes:23437688605 (23.4 GB)
        Interrupt:20 Memory:fe500000-fe520000
```

A saída do comando `ifconfig` inclui vários detalhes tanto da camada de Internet quanto da camada física. (Às vezes, ela não inclui nem mesmo um endereço de Internet!) Discutiremos a saída com mais detalhes posteriormente, mas, por enquanto, concentre-se na segunda linha, que informa que o host está configurado para ter um endereço IPv4 (`inet addr`) igual a 10.23.2.4. Na mesma linha, um `Mask` é informado como sendo 255.255.255.0. Essa é a *máscara de sub-rede*, que define a sub-rede à qual um endereço IP pertence. Vamos ver como ela funciona.

 **Observação:** o comando `ifconfig`, assim como outros que você verá mais adiante neste capítulo (como `route` e `arp`), foram tecnicamente suplantados pelo comando `ip` mais recente. O comando `ip` pode fazer mais do que os comandos antigos, e é preferível ao escrever scripts. No entanto a maioria das pessoas ainda usa os comandos antigos quando trabalha manualmente com a rede, e esses comandos também podem ser usados em outras versões de Unix. Por esse motivo, usaremos os comandos de estilo antigo.

9.3.2 Sub-redes

Uma *sub-rede* é um grupo de hosts conectados, com endereços IP em algum tipo de ordem. Normalmente, os hosts estão na mesma rede física, como mostrado na figura 9.2. Por exemplo, os hosts entre 10.23.2.1 e 10.23.2.254 poderiam formar uma sub-rede, assim como todos os hosts entre 10.23.1.1 e 10.23.255.254.

Uma sub-rede é definida por meio de duas partes: um *prefixo de rede* e uma máscara de sub-rede (por exemplo, aquela que está na saída do comando `ifconfig` na seção anterior). Vamos supor que você queira criar uma sub-rede contendo os endereços IP entre 10.23.2.1 e 10.23.2.254. O prefixo de rede é a parte *comum* a todos os endereços da sub-rede; nesse exemplo, é 10.23.2.0, e a máscara de sub-rede é 255.255.255.0. Vamos

ver por que esses são os números corretos.

Não está imediatamente claro como o prefixo e a máscara funcionam em conjunto para fornecer todos os endereços IP possíveis em uma sub-rede. Observar os números em formato binário ajuda a esclarecer essa situação. A máscara marca as localizações dos bits em um endereço IP que são comuns à sub-rede. Por exemplo, a seguir, apresentamos os formatos binários de 10.23.2.0 e de 255.255.255.0:

```
10.23.2.0:      00001010 00010111 00000010 00000000
255.255.255.0:  11111111 11111111 11111111 00000000
```

Vamos agora usar negrito para marcar as localizações dos bits em 10.23.2.0 que sejam iguais a 1 em 255.255.255.0:

```
10.23.2.0: 00001010 00010111 00000010 00000000
```

Observe os bits que *não* estão em negrito. Você pode definir qualquer quantidade desses bits com 1 para obter um endereço IP válido nessa sub-rede, com exceção de tudo 0 ou tudo 1.

Reunindo todas as informações, você pode ver como um host com um endereço IP igual a 10.23.2.1 e uma máscara de sub-rede igual a 255.255.255.0 estão na mesma sub-rede que qualquer outro computador que tenha endereços IP começados com 10.23.2. Essa sub-rede como um todo pode ser representada como 10.23.2.0/255.255.255.0.

9.3.3 Máscaras de sub-rede comuns e notação CIDR

Se estiver com sorte, você lidará somente com máscaras de sub-rede simples como 255.255.255.0 ou 255.255.0.0, porém você pode ter azar e encontrar algo como 255.255.255.192, em que não é tão simples determinar o conjunto de endereços que pertencem à sub-rede. Além do mais, é mais provável que você também encontre uma forma diferente de representação de sub-redes chamada notação *CIDR* (Classless Inter-Domain Routing), em que uma sub-rede como 10.23.2.0/255.255.255.0 é representada como 10.23.2.0/24.

Para entender o que isso significa, dê uma olhada na máscara em formato binário (como no exemplo que vimos na seção anterior). Você perceberá que quase todas as máscaras de sub-rede são apenas um conjunto de 1s seguido de um conjunto de 0s. Por exemplo, você acabou de ver que 255.255.255.0 em formato binário corresponde a 24 bits iguais a 1 seguidos de 8 bits iguais a 0. A notação CIDR identifica a máscara de sub-rede pela quantidade de 1s *iniciais* na máscara de sub-rede. Desse modo, uma combinação como 10.23.2.0/24 inclui tanto o prefixo da sub-rede quanto a sua máscara.

A tabela 9.1 mostra vários exemplos de máscaras de sub-rede e seus formatos CIDR.

Tabela 9.1 – Máscaras de sub-rede

Formato longo	Formato CIDR
255.0.0.0	8
255.255.0.0	16
255.240.0.0	12
255.255.255.0	24
255.255.255.192	26

 **Observação:** se você não estiver familiarizado com a conversão entre os formatos decimal, binário e hexadecimal, um utilitário de calculadora como `bc` ou `dc` poderá ser usado para fazer a conversão entre as representações em diferentes bases. Por exemplo, no `bc`, você pode executar o comando `obase=2; 240` para exibir o número 240 em formato binário (base 2).

Identificar sub-redes e seus hosts é o primeiro bloco de construção para entender como a Internet funciona. No entanto é preciso ainda conectar as sub-redes.

9.4 Rotas e a tabela de roteamento do kernel

Conectar sub-redes de Internet é um processo que implica principalmente identificar os hosts conectados a mais de uma sub-rede. Retornando à figura 9.2, pense no Host A no endereço IP 10.23.2.4. Esse host está conectado a uma rede local 10.23.2.0/24 e pode alcançar diretamente os hosts dessa rede. Para alcançar os hosts no restante da Internet, ele deverá se comunicar por meio do roteador em 10.23.2.1.

Como o kernel do Linux faz a distinção entre esses dois tipos diferentes de destino? Ele usa uma configuração de destinos chamada de *tabela de roteamento* para determinar seu comportamento quanto ao roteamento. Para mostrar a tabela de roteamento, utilize o comando `route -n`. A seguir, apresentamos o que você poderá ver no caso de um host simples como 10.23.2.4:

```
$ route -n
```

```
Kernel IP routing table
```

```

Destination    Gateway      Genmask      Flags Metric Ref  Use Iface
0.0.0.0        10.23.2.1   0.0.0.0      UG  0    0    0 eth0
10.23.2.0      0.0.0.0     255.255.255.0 U    1    0    0 eth0
```

As duas últimas linhas, nesse caso, contêm as informações de roteamento. A coluna `Destination` contém um prefixo de rede, e a coluna `Genmask` é a máscara de rede correspondente a essa rede. Há duas redes definidas nessa saída: 0.0.0.0/0 (que corresponde a todos os endereços da Internet) e 10.23.2.0/24. Cada rede tem um U na

coluna `Flags`, indicando que a rota está ativa (“up”).

Os destinos diferem na combinação de suas colunas `Gateway` e `Flags`. Para `0.0.0.0/0`, há um `G` na coluna `Flags`, que significa que a comunicação para essa rede deve ser enviada por meio do gateway na coluna `Gateway` (`10.23.2.1`, nesse caso). Entretanto, para `10.23.2.0/24`, não há nenhum `G` em `Flags`, indicando que a rede está diretamente conectada de alguma maneira. Nesse caso, `0.0.0.0` é usado como valor substituto em `Gateway`. Ignore as demais colunas da saída, por enquanto.

Há um detalhe complicado: suponha que o host queira enviar algo para `10.23.2.132`, que corresponde a ambas as regras, `0.0.0.0/0` e `10.23.2.0/24`, na tabela de roteamento. Como o kernel sabe que deve usar a segunda? Ele escolhe o prefixo de destino mais longo que faça a correspondência. É nesse caso que o formato de rede CIDR vem particularmente a calhar: `10.23.2.0/24` faz a correspondência, e seu prefixo tem 24 bits de comprimento; `0.0.0.0/0` também faz a correspondência, porém seu prefixo tem tamanho 0 (ou seja, ele não tem prefixo), portanto a regra para `10.23.2.0/24` tem prioridade.

👍 **Observação:** a opção `-n` diz a `route` para mostrar endereços IP em vez de mostrar os hosts e as redes pelo nome. Essa é uma opção importante a ser lembrada porque você poderá usá-la em outros comandos relacionados a redes, por exemplo, em `netstat`.

9.4.1 O gateway default

Uma entrada para `0.0.0.0/0` na tabela de roteamento tem um significado especial, pois ela faz a correspondência com qualquer endereço da Internet. Essa é a *rota default*, e o endereço configurado na coluna `Gateway` (na saída de `route -n`) na rota default é o *gateway default*. Quando não houver nenhuma outra regra que faça a correspondência, a rota default sempre o fará, e o gateway default é para onde você enviará as mensagens quando não houver nenhuma outra opção. Você pode configurar um host sem um gateway default, porém ele não será capaz de alcançar hosts diferentes dos destinos que estiverem na tabela de roteamento.

👍 **Observação:** na maioria das redes com uma máscara de rede igual a `255.255.255.0`, o roteador geralmente estará no endereço 1 da sub-rede (por exemplo, `10.23.2.1` em `10.23.2.0/24`). Como isso é apenas uma convenção, pode haver exceções.

9.5 Ferramentas básicas para ICMP e DNS

Agora é hora de dar uma olhada em alguns utilitários básicos práticos para ajudar você a interagir com os hosts. Essas ferramentas usam dois protocolos particularmente interessantes: o ICMP (Internet Control Message Protocol), que pode ajudar a

identificar problemas com conectividade e roteamento, e o sistema DNS (Domain Name Service), que mapeia nomes a endereços IP para que você não precise se lembrar de um conjunto de números.

9.5.1 ping

O ping (acesse <http://ftp.arl.mil/~mike/ping.html>) é uma das ferramentas de debugging de rede mais básicas. Ele envia pacotes de solicitação de eco ICMP para um host, pedindo a um host receptor que retorne o pacote a quem enviou. Se o host receptor receber o pacote e estiver configurado para responder, ele enviará de volta um pacote de resposta de eco ICMP.

Por exemplo, suponha que você tenha executado `ping 10.23.2.1` e que tenha obtido o resultado a seguir:

```
$ ping 10.23.2.1
```

```
PING 10.23.2.1 (10.23.2.1) 56(84) bytes of data.
```

```
64 bytes from 10.23.2.1: icmp_req=1 ttl=64 time=1.76 ms
```

```
64 bytes from 10.23.2.1: icmp_req=2 ttl=64 time=2.35 ms
```

```
64 bytes from 10.23.2.1: icmp_req=4 ttl=64 time=1.69 ms
```

```
64 bytes from 10.23.2.1: icmp_req=5 ttl=64 time=1.61 ms
```

A primeira linha informa que você está enviando pacotes de 56 bytes (84 bytes, se os cabeçalhos forem incluídos) para 10.23.2.1 (por padrão, um pacote por segundo), e as linhas restantes representam respostas de 10.23.2.1. As partes mais importantes da saída são o número de sequência (`icmp_req`) e o tempo de ida e volta (`time`). O número de bytes retornados corresponde ao tamanho do pacote enviado mais oito. (O conteúdo dos pacotes não é importante para você.)

Um intervalo nos números de sequência, como aquele entre 2 e 4, normalmente significa que há algum tipo de problema de conectividade. É possível que os pacotes cheguem fora de ordem e, caso isso aconteça, há algum tipo de problema, pois o ping envia somente um pacote por segundo. Se uma resposta demorar mais de um segundo (1000 ms) para chegar, a conexão estará extremamente lenta.

O tempo de ida e volta corresponde ao tempo total decorrido entre o momento em que o pacote de solicitação sai até o momento em que o pacote de resposta chega. Se não houver nenhuma maneira de alcançar o destino, o último roteador a ver o pacote retornará um pacote ICMP “host unreachable” (host inalcançável) ao ping.

Em uma LAN com fio, você não deve esperar absolutamente nenhuma perda de pacotes e deverá obter números bem baixos para o tempo de ida e volta. (A saída do exemplo anterior é de uma rede wireless.) Você também não deve esperar ter nenhuma perda de

pacotes de sua rede para o ISP e vice-versa, além de obter tempos razoavelmente uniformes de ida e volta.

👍 **Observação:** por questões de segurança, nem todos os hosts na Internet respondem a pacotes de solicitação de eco ICMP, portanto você poderá perceber que será possível se conectar a um site em um host, porém não obterá resposta para um ping.

9.5.2 traceroute

O programa traceroute baseado em ICMP será prático quando você entrar em contato com o material sobre roteamento, mais adiante neste capítulo. Use `traceroute host` para ver o caminho que seus pacotes percorrem até um host remoto. (`traceroute -n host` desabilitará a pesquisa de nomes de host.)

Um dos melhores recursos do traceroute é que ele informa os tempos de retorno em cada passo na rota, conforme mostrado neste fragmento de saída:

```
4 206.220.243.106 1.163 ms 0.997 ms 1.182 ms
5 4.24.203.65 1.312 ms 1.12 ms 1.463 ms
6 64.159.1.225 1.421 ms 1.37 ms 1.347 ms
7 64.159.1.38 55.642 ms 55.625 ms 55.663 ms
8 209.247.10.230 55.89 ms 55.617 ms 55.964 ms
9 209.244.14.226 55.851 ms 55.726 ms 55.832 ms
10 209.246.29.174 56.419 ms 56.44 ms 56.423 ms
```

Como essa saída mostra um salto enorme de latência entre os hops 6 e 7, essa parte da rota provavelmente é algum tipo de link de longa distância.

A saída do traceroute pode ser inconsistente. Por exemplo, as respostas podem sofrer timeout em um determinado passo somente para “reaparecer” em passos posteriores. O motivo normalmente está no fato de o roteador nesse passo ter se recusado a retornar a saída de debugging que o traceroute quer, porém os roteadores em passos posteriores ficaram felizes em retornar a saída. Além do mais, um roteador pode optar por atribuir uma prioridade mais baixa para o tráfego de debugging em relação ao tráfego normal.

9.5.3 DNS e host

Os endereços IP são difíceis de lembrar e estão sujeitos à mudança, motivo pelo qual normalmente usamos nomes como *www.example.com* em seu lugar. A biblioteca DNS em seu sistema geralmente cuida dessa tradução automaticamente, porém talvez você queira, às vezes, fazer a tradução manualmente entre um nome e um endereço IP. Para descobrir o endereço IP por trás de um nome de domínio, utilize o comando `host`:

```
$ host www.example.com
```

```
www.example.com has address 93.184.216.119
```

Observe como esse exemplo tem tanto o endereço IPv4 93.184.216.119 quanto o endereço IPv6 mais longo. Isso significa que esse host também tem um endereço na versão de próxima geração da Internet.

O host também pode ser usado de modo reverso: forneça um endereço IP no lugar de um nome de host para tentar descobrir o nome por trás do endereço IP. Mas não espere que isso funcione de maneira confiável. Muitos nomes de host podem representar um único endereço IP, e o DNS não sabe como determinar qual nome de host deve corresponder a esse endereço. O administrador do domínio deve configurar essa pesquisa reversa manualmente e, com frequência, ele não o faz. (Há muito mais a ser discutido sobre o DNS além do comando `host`. Discutiremos a configuração básica do cliente mais adiante, na seção 9.12).

9.6 A camada física e a Ethernet

Um dos pontos mais importantes a entender sobre a Internet é que ela é uma rede de *software*. Nada do que discutimos até agora é específico de um hardware e, na realidade, um dos motivos do sucesso da Internet é que ela funciona em quase todos os tipos de computadores, sistemas operacionais e redes físicas. No entanto continua sendo necessário colocar uma camada de rede sobre algum tipo de hardware, e essa interface é chamada de camada física.

Neste livro, daremos uma olhada no tipo mais comum de camada física: uma rede Ethernet. A família IEEE 802 de padrões de documentos define vários tipos diferentes de redes Ethernet, desde redes com fio a redes sem fio, porém todas elas têm alguns aspectos em comum, em particular:

- Todos os dispositivos em uma rede Ethernet têm um *endereço MAC* (Media Access Control), às vezes chamado de *endereço de hardware*. Esse endereço é independente do endereço IP de um host e é único na rede Ethernet do host (porém não necessariamente em uma rede de software mais ampla, como a Internet). Um exemplo de endereço MAC é 10:78:d2:eb:76:97.
- Os dispositivos em uma rede Ethernet enviam mensagens em *frames*, que são wrappers (encapsuladores) em torno dos dados enviados. Um frame contém os endereços MAC de origem e de destino.

A Ethernet não tenta realmente ir além do hardware em uma única rede. Por exemplo, se você tiver duas redes Ethernet diferentes, com um host conectado a ambas as redes (e dois dispositivos de interface de rede diferentes), não será possível transmitir um frame

diretamente de uma rede Ethernet para a outra, a menos que você instale um bridge Ethernet especial. E é nesse ponto que as camadas de rede mais altas (como a camada de Internet) entram em cena. Por convenção, cada rede Ethernet normalmente também é uma sub-rede da Internet. Mesmo que um frame não possa sair de uma rede física, um roteador pode extrair os dados de um frame, empacotá-los novamente e enviá-los a um host em uma rede física diferente, que é exatamente o que acontece na Internet.

9.7 Entendendo as interfaces de rede do kernel

As camadas física e de Internet devem estar conectadas de maneira que permitam à camada de Internet manter sua flexibilidade quanto à independência do hardware. O kernel do Linux mantém sua própria divisão entre as duas camadas e provê padrões de comunicação para ligá-las, que se chamam *interface de rede* (do kernel). Ao configurar uma interface de rede, efetuamos a ligação entre as configurações de endereço IP do lado da Internet e a identificação de hardware do lado do dispositivo físico. As interfaces de rede têm nomes que normalmente indicam o tipo de hardware subjacente, como *eth0* (a primeira placa Ethernet do computador) e *wlan0* (uma interface wireless).

Na seção 9.3.1, você conheceu o comando mais importante para visualizar ou configurar manualmente os parâmetros da interface de rede: o `ifconfig`. Lembre-se da saída a seguir:

```
eth0  Link encap:Ethernet HWaddr 10:78:d2:eb:76:97
      inet addr:10.23.2.4 Bcast:10.23.2.255 Mask:255.255.255.0
      inet6 addr: fe80::1278:d2ff:feeb:7697/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
      RX packets:85076006 errors:0 dropped:0 overruns:0 frame:0
      TX packets:68347795 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:86427623613 (86.4 GB) TX bytes:23437688605 (23.4 GB)
      Interrupt:20 Memory:fe500000-fe520000
```

Para cada interface de rede, o lado esquerdo da saída mostra o nome da interface, e o lado direito contém configurações e estatísticas da interface. Além das partes referentes à camada de Internet que já discutimos, você também pode ver o endereço MAC da camada física (HWaddr). As linhas contendo UP e RUNNING informam que a interface está funcionando.

Embora `ifconfig` mostre algumas informações de hardware (nesse caso, até mesmo algumas configurações de baixo nível do dispositivo, como a interrupção e a memória usadas), ele foi concebido principalmente para visualizar e configurar as camadas de software associadas às interfaces. Para explorar mais o hardware e a camada física por

trás de uma interface de rede, utilize algo como o comando `ethtool` para exibir ou para alterar as configurações das placas Ethernet. (Daremos uma olhada rapidamente em redes wireless na seção 9.23.)

9.8 Introdução à configuração das interfaces de rede

Você já viu todos os elementos básicos envolvidos nos níveis mais baixos de uma pilha de rede: a camada física, a camada de rede (de Internet) e as interfaces de rede do kernel do Linux. Para combinar essas partes de modo a conectar um computador Linux à Internet, você ou um software deverão fazer o seguinte:

1. Conectar o hardware de rede e garantir que o kernel tenha um driver para ele. Se o driver esteve presente, `ifconfig -a` exibirá uma interface de rede do kernel correspondente ao hardware.
2. Executar qualquer configuração adicional da camada física, por exemplo, selecionar um nome de rede ou uma senha.
3. Associar um endereço IP e uma máscara de rede à interface de rede do kernel para que os device drivers do kernel (camada física) e os subsistemas de Internet (camada de Internet) possam conversar um com o outro.
4. Acrescentar quaisquer rotas adicionais necessárias, incluindo o gateway default.

Quando todos os computadores eram grandes caixas estáticas interligadas, isso era relativamente simples: o kernel executava o passo 1, o passo 2 não era necessário, o passo 3 era executado por meio do comando `ifconfig` e o passo 4, com o comando `route`.

Para definir manualmente o endereço IP e a máscara de rede para uma interface de rede do kernel, você executava:

```
# ifconfig interface endereço netmask máscara
```

Nesse caso, *interface* é o nome da interface, por exemplo `eth0`. Quando a interface estivesse ativa, você estaria pronto para adicionar as rotas, o que, normalmente, era uma simples questão de definir o gateway default, desta maneira:

```
# route add default gw endereço-gw
```

O parâmetro *endereço-gw* corresponde ao endereço IP de seu gateway default; ele *deve* ser um endereço em uma sub-rede conectada localmente, definido pelos parâmetros *endereço* e *máscara* de uma de suas interfaces de rede.

9.8.1 Adicionando e apagando rotas manualmente

Para remover um gateway default, execute:

```
# route del -net default
```

Podemos facilmente sobrepor o gateway default com outras rotas. Por exemplo, suponha que seu computador esteja na sub-rede 10.23.2.0/24, você queira alcançar uma sub-rede em 192.168.45.0/24 e saiba que 10.23.2.44 pode atuar como um roteador para essa sub-rede. Execute o comando a seguir para enviar o tráfego destinado a 192.168.45.0 para aquele roteador:

```
# route add -net 192.168.45.0/24 gw 10.23.2.44
```

Não é necessário especificar o roteador para apagar uma rota:

```
# route del -net 192.168.45.0/24
```

Porém, antes de enlouquecer com as rotas, saiba que criar confusão com rotas geralmente é mais complicado do que parece. Nesse exemplo em particular, devemos também garantir que o roteamento para todos os hosts em 192.163.45.0/24 possa levar de volta a 10.23.2.0/24, ou a primeira rota que você adicionar basicamente será inútil.

Normalmente, devemos manter o máximo de simplicidade aos clientes, configurando as redes de modo que seus hosts precisem somente de uma rota default. Se você precisar de várias sub-redes e da capacidade de efetuar roteamento entre elas, geralmente é melhor configurar os roteadores para que atuem como gateways default para que façam todo o trabalho de roteamento entre as diferentes sub-redes locais. (Você verá um exemplo na seção 9.17.)

9.9 Configuração de rede ativada no boot

Discutimos maneiras de configurar uma rede manualmente, e o modo tradicional de garantir que a configuração de rede de um computador estivesse correta era fazer o init executar um script para ativar a configuração manual no momento do boot. Isso se resume a executar ferramentas como `ifconfig` e `route` em algum ponto da cadeia de eventos de boot. Muitos servidores continuam fazendo isso dessa maneira.

Houve várias tentativas no Linux de padronizar os arquivos de configuração para as redes no momento do boot. As ferramentas `ifup` e `ifdown` fazem isso – por exemplo, um script de boot (teoricamente) pode executar `ifup eth0` para executar os comandos `ifconfig` e `route` corretos para a interface `eth0`. Infelizmente, diferentes distribuições têm implementações totalmente distintas de `ifup` e `ifdown` e, como resultado, seus arquivos de configuração são também completamente diferentes. O Ubuntu, por exemplo, usa o pacote `ifupdown` com arquivos de configuração em `/etc/network`, e o Fedora usa seu próprio conjunto de scripts com a configuração em `/etc/sysconfig/network-scripts`.

Não é preciso conhecer os detalhes desses arquivos de configuração, mas se você

insistir em fazer tudo manualmente e ignorar as ferramentas de configuração de sua distribuição, basta consultar os formatos nas páginas de manual como `ifup(8)` e `interfaces(5)`. Porém é importante saber que esse tipo de configuração ativada no boot geralmente não é nem mesmo usada. Com mais frequência, você a verá somente para a interface de rede localhost (ou `lo`; veja a seção 9.13), porque ela é inflexível demais para atender às necessidades dos sistemas modernos.

9.10 Problemas com a configuração manual de rede ativada no boot

Embora a maioria dos sistemas costumasse configurar a rede em seus sistemas de boot – e muitos servidores continuam fazendo isso –, a natureza dinâmica das redes modernas resultou no fato de a maioria dos computadores não ter endereços IP estáticos (imutáveis). Em vez de armazenar o endereço IP e outras informações de rede em seu computador, esse obterá essas informações de outro lugar da rede física local quando ele se conectar a essa rede pela primeira vez. A maioria das aplicações de rede clientes normais não se importa particularmente com o endereço IP usado pelo seu computador, desde que ele funcione. As ferramentas DHCP (Dynamic Host Configuration Protocol, ou Protocolo de configuração dinâmica de hosts, descrito na seção 9.16) realizam a configuração básica da camada de rede nos clientes comuns.

No entanto há mais detalhes nessa história. Por exemplo, as redes wireless acrescentam novas dimensões à configuração da interface, por exemplo, nomes de rede, autenticação e técnicas de criptografia. Ao dar um passo para trás e contemplar o quadro geral, você verá que o seu sistema precisa ter uma maneira de responder às seguintes perguntas:

- Se o computador tiver várias interfaces de rede físicas (por exemplo, um notebook com Ethernet com e sem fio), como você escolherá qual(is) delas deverá usar?
- Como o computador deverá configurar a interface física? Para redes wireless, isso inclui efetuar scanning em busca dos nomes das redes, selecionar um nome e negociar a autenticação.
- Depois que a interface de rede física estiver conectada, como o computador deverá configurar as camadas de rede de software, por exemplo, a camada de Internet?
- Como você pode deixar que um usuário selecione as opções de conectividade? Por exemplo, como você permitirá que um usuário escolha uma rede wireless?
- O que o computador deverá fazer se ele perder a conectividade em uma interface de rede?

Responder a essas perguntas normalmente representa mais trabalho do que aquele com que os scripts simples de boot conseguem lidar, e é realmente muito complicado fazer tudo manualmente. A resposta é usar um serviço de sistema que possa monitorar as redes físicas e escolher (e configurar automaticamente) as interfaces de rede do kernel de acordo com um conjunto de regras que façam sentido ao usuário. O serviço também deverá ser capaz de responder às solicitações dos usuários, que deverão poder alterar a rede wireless em que estiverem, sem precisar se tornar root somente para ajustar as configurações de rede sempre que algo mudar.

9.11 Gerenciadores de configuração de rede

Há diversas maneiras de configurar redes automaticamente em sistemas baseados em Linux. A opção mais amplamente usada nos desktops e nos notebooks é o NetworkManager. Outros sistemas de gerenciamento de configuração de rede são voltados principalmente para sistemas embarcados menores, como o `netifd` do OpenWRT, o serviço ConnectivityManager do Android, o ConnMan e o Wicd. Discutiremos brevemente o NetworkManager porque é o sistema que você terá mais chances de encontrar. Porém não forneceremos uma quantidade imensa de detalhes, pois, depois que você observar o quadro geral, o NetworkManager e outros sistemas de configuração serão mais transparentes.

9.11.1 Operação do NetworkManager

O NetworkManager é um daemon que o sistema inicia no boot. Como todos os daemons, ele não depende de um componente de desktop em execução. Sua tarefa é ouvir eventos do sistema e de usuários e mudar a configuração de rede de acordo com um conjunto de regras.

Quando está executando, o NetworkManager mantém dois níveis básicos de configuração. O primeiro é um conjunto de informações sobre os dispositivos de hardware disponíveis, que normalmente é obtido do kernel e mantido ao monitorar o `udev` sobre o Desktop Bus (D-Bus). O segundo nível de configuração é uma lista mais específica de *conexões*: dispositivos de hardware e parâmetros de configuração adicionais das camadas física e de rede. Por exemplo, uma rede wireless pode ser representada como uma conexão.

Para ativar uma conexão, o NetworkManager geralmente delega as tarefas a outras ferramentas e daemons especializados de rede, como o `dhclient`, para obter a configuração da camada de Internet de uma rede física conectada localmente. Como as ferramentas e

os esquemas de configuração de rede variam entre as distribuições, o NetworkManager utiliza plugins para fazer a interface com elas, em vez de impor o seu próprio padrão. Há plugins tanto para a interface de configuração de estilo Debian/Ubuntu quanto para Red Hat, por exemplo.

Na inicialização, o NetworkManager reúne todas as informações de dispositivos de rede disponíveis, pesquisa sua lista de conexões e então decide tentar ativar uma delas. A seguir, descrevemos como ele toma essa decisão para interfaces Ethernet:

1. Se uma conexão com fio estiver disponível, tenta conectar utilizando essa conexão. Caso contrário, tenta usar as conexões wireless.
2. Pesquisa a lista de redes wireless disponíveis. Se uma rede com a qual você já se conectou anteriormente estiver disponível, o NetworkManager tentará se conectar com ela novamente.
3. Se houver mais de uma rede wireless disponível com as quais já houve uma conexão anterior, a rede com a conexão mais recente será selecionada.

Após estabelecer uma conexão, o NetworkManager a manterá até que a conexão seja perdida, uma rede melhor se torne disponível (por exemplo, um cabo de rede seja conectado enquanto você estiver conectado via wireless) ou o usuário force uma mudança.

9.11.2 Interagindo com o NetworkManager

A maioria dos usuários interage com o NetworkManager por meio de um applet do desktop – normalmente, é um ícone no canto superior ou inferior à direita que indica o status da conexão (com fio, sem fio ou desconectado). Ao clicar no ícone, você obterá várias opções de conectividade, por exemplo, várias redes wireless entre as quais escolher e uma opção para se desconectar da rede atual. Cada ambiente de desktop tem sua própria versão desse applet, portanto ele terá uma aparência um pouco diferente para cada um.

Além do applet, há algumas ferramentas que você pode usar para consultar e controlar o NetworkManager a partir de seu shell. Para ter um resumo bem rápido do status de sua conexão atual, utilize o comando `nm-tool` sem argumentos. Você obterá uma lista das interfaces e os parâmetros de configuração. Em alguns aspectos, é como `ifconfig`, exceto pelo fato de haver mais detalhes, especialmente quando conexões wireless estiverem sendo visualizadas.

Para controlar o NetworkManager a partir da linha de comando, utilize o comando `nmcli`. Esse é um comando, de certo modo, extenso. Consulte a página de manual `nmcli(1)` para

obter mais informações.

Por fim, o utilitário `nm-online` informará se a rede está ou não ativa. Se a rede estiver ativa, o comando retornará zero como seu código de saída; esse valor será diferente de zero, caso contrário. (Para saber mais sobre como usar um código de saída em um shell script, veja o capítulo 11.)

9.11.3 Configuração do NetworkManager

O diretório de configuração geral do NetworkManager normalmente é */etc/NetworkManager*, e há vários tipos diferentes de configuração. O arquivo geral de configuração é *NetworkManager.conf*. O formato é semelhante ao de arquivos *.desktop* de estilo XDG e de arquivos Microsoft *.ini*, com parâmetros chave-valor que se enquadram em diferentes seções. Você perceberá que quase todo arquivo de configuração tem uma seção `[main]` que define os plugins a serem usados. A seguir, vemos um exemplo simples que ativa o plugin `ifupdown` usado pelo Ubuntu e pelo Debian:

```
[main]
plugins=ifupdown,keyfile
```

Outros plugins específicos de distribuição são o `ifcfg-rh` (para distribuições de estilo Red Hat) e o `ifcfg-suse` (para SuSE). O plugin `keyfile` que você também pode ver nesse caso suporta o arquivo de configuração nativo do NetworkManager. Ao usar o plugin, você poderá ver as conexões conhecidas pelo sistema em */etc/NetworkManager/system-connections*.

Na maioria das vezes, não será necessário alterar *NetworkManager.conf*, pois as opções mais específicas de configuração se encontram em outros arquivos.

Interfaces não administradas

Embora você possa querer que o NetworkManager gerencie a maioria de suas interfaces de rede, talvez haja ocasiões em que você queira que ele ignore as interfaces. Por exemplo, não há nenhum motivo pelo qual a maioria dos usuários vá precisar de qualquer tipo de configuração dinâmica na interface `localhost (lo)`, pois a configuração não mudará nunca. Você também vai querer configurar essa interface bem cedo no processo de boot porque serviços básicos do sistema, com frequência, dependem dela. A maioria das distribuições mantém o NetworkManager longe do `localhost`.

Você pode dizer ao NetworkManager para desconsiderar uma interface usando plugins. Se você estiver usando o plugin `ifupdown` (por exemplo, no Ubuntu e no Debian), adicione a configuração da interface ao seu arquivo */etc/network/interfaces* e, em

seguida, configure o valor de `managed` com `false` na seção `ifupdown` do arquivo *NetworkManager.conf*:

```
[ifupdown]
managed=false
```

Para o plugin `ifcfg-rh` usado pelo Fedora e pelo Red Hat, procure uma linha como a que se segue no diretório */etc/sysconfig/network-scripts* que contém os arquivos de configuração *ifcfg-**:

```
NM_CONTROLLED=yes
```

Se essa linha não estiver presente ou se o valor estiver configurado com `no`, o NetworkManager irá ignorar a interface. Por exemplo, você encontrará esse valor desabilitado no arquivo *ifcfg-lo*. Também é possível especificar um endereço de hardware a ser ignorado, da seguinte maneira:

```
HWADDR=10:78:d2:eb:76:97
```

Se nenhum desses esquemas de configuração de rede for usado, ainda será possível usar o plugin `keyfile` para especificar diretamente os dispositivos não administrados em seu arquivo *NetworkManager.conf* usando o endereço MAC. Aqui está o modo como isso se parecerá:

```
[keyfile]
unmanaged-devices=mac:10:78:d2:eb:76:97;mac:1c:65:9d:cc:ff:b9
```

Dispatching

Um último detalhe da configuração do NetworkManager está relacionado à especificação de ações adicionais de sistema, executadas quando uma interface de rede é ativada ou desativada. Por exemplo, alguns daemons de rede devem saber quando devem começar ou parar de ouvir uma interface para que funcionem corretamente (por exemplo, o daemon `secure shell` discutido no próximo capítulo).

Quando o status da interface de rede em um sistema mudar, o NetworkManager executará tudo o que estiver em */etc/NetworkManager/dispatcher.d* com um argumento como `up` ou `down`. Isso é relativamente simples, porém muitas distribuições têm seus próprios scripts de controle de rede, de modo que eles não colocam os scripts individuais de dispatcher nesse diretório. O Ubuntu, por exemplo, tem somente um script chamado `01ifupdown` que executa tudo o que estiver em um subdiretório apropriado de */etc/network*, por exemplo, */etc/network/if-up.d*.

Quanto ao restante da configuração do NetworkManager, os detalhes desses scripts não são relativamente importantes; tudo o que você deve saber é como descobrir a localização apropriada caso seja necessário fazer alguma adição ou alteração. Como

sempre, não tenha medo de olhar os scripts em seu sistema.

9.12 Resolvendo nomes de hosts

Uma das últimas tarefas básicas em qualquer configuração de rede é a resolução de nomes de hosts com o DNS. Você já conheceu a ferramenta de resolução `host`, que traduz um nome como *www.example.com* para um endereço IP como 10.23.2.132.

O DNS difere dos elementos de rede que vimos até agora porque ele está na camada de aplicação, totalmente no espaço de usuário. Tecnicamente, ele está um pouco fora de lugar neste capítulo, ao lado da discussão sobre as camadas física e de Internet, porém, sem uma configuração apropriada de DNS, sua conexão com a Internet praticamente será inútil. Ninguém em sã consciência divulga endereços IP para sites e para endereços de email porque o endereço IP de um host está sujeito a mudanças e não é fácil se lembrar de um conjunto de números. Os serviços de configuração automática de rede como o DHCP quase sempre incluem a configuração de DNS.

Quase todas as aplicações de rede em um sistema Linux realizam pesquisas de DNS. O processo de resolução normalmente se desdobra nos seguintes passos:

1. A aplicação chama uma função para pesquisar o endereço IP por trás de um nome de host. Essa função está na biblioteca compartilhada do sistema, portanto a aplicação não precisa conhecer os detalhes sobre como ela funciona nem saber se a implementação será alterada.
2. Quando a função da biblioteca compartilhada for executada, ela atuará de acordo com um conjunto de regras (encontrado em */etc/nsswitch.conf*) para determinar um plano de ação para as pesquisas. Por exemplo, as regras normalmente dizem que, mesmo antes de acessar o DNS, é preciso saber se há alguma sobreposição manual da regra no arquivo */etc/hosts*.
3. Quando a função decide usar o DNS para pesquisar o nome, um arquivo de configuração adicional é consultado para encontrar um servidor de nomes DNS. O servidor de nomes é especificado como um endereço IP.
4. A função envia uma solicitação de pesquisa ao DNS (pela rede) para o servidor de nomes.
5. O servidor de nomes responde com o endereço IP referente ao nome do host, e a função retorna esse endereço IP à aplicação.

Essa é a versão simplificada. Em um sistema moderno normal, há mais atores tentando agilizar a transação e/ou adicionar flexibilidade. Vamos ignorar isso por enquanto e dar

uma olhada com mais detalhes nas partes básicas.

9.12.1 /etc/hosts

Na maioria dos sistemas, o arquivo */etc/hosts* pode se sobrepor às pesquisas de nomes de host. Em geral, esse arquivo tem o seguinte aspecto:

```
127.0.0.1    localhost
10.23.2.3    atlantic.aem7.net    atlantic
10.23.2.4    pacific.aem7.net    pacific
```

Quase sempre você verá a entrada para localhost nesse arquivo (consulte a seção 9.13).

 **Observação:** nos velhos tempos difíceis, havia um arquivo de hosts central que todos copiavam para seus próprios computadores para permanecerem atualizados (veja as RFCs 606, 608, 623 e 625), porém, à medida que a ARPANET/Internet cresceu, isso rapidamente deixou de ser prático.

9.12.2 resolv.conf

O arquivo tradicional de configuração para servidores DNS é */etc/resolv.conf*. Quando tudo era mais simples, um exemplo típico poderia ter o aspecto apresentado a seguir, em que os endereços do servidor de nomes do ISP eram 10.32.45.23 e 10.3.2.3:

```
search mydomain.example.com example.com
nameserver 10.32.45.23
nameserver 10.3.2.3
```

A linha *search* define regras para nomes de host incompletos (somente a primeira parte do nome do host; por exemplo, *myserver* no lugar de *myserver.example.com*). Nesse caso, a biblioteca *resolver* tentaria pesquisar *host.mydomain.example.com* e *host.example.com*. Porém, em geral, a situação não é mais tão simples assim. Muitas melhorias e modificações foram feitas à configuração do DNS.

9.12.3 Caching e DNS sem configuração

Há dois problemas principais com a configuração tradicional de DNS. Em primeiro lugar, o computador local não faz cache das respostas do servidor de nomes, portanto, acessos frequentes e repetitivos à rede poderão ser desnecessariamente lentos por causa de solicitações ao servidor de nomes. Para resolver esse problema, muitos computadores (e roteadores, se estiverem atuando como servidores de nomes) executam um daemon intermediário para interceptar as solicitações aos servidores de nomes e retornam uma resposta de cache para as solicitações aos serviços de nome, se for possível; caso contrário, as solicitações são enviadas para um verdadeiro servidor de nomes. Dois desses daemons mais comuns para o Linux são *dnsmasq* e *nsd*. Também

podemos instalar o BIND (o daemon de servidor de nomes padrão do Unix) como cache. Com frequência, você poderá dizer se está executando um daemon de caching para servidor de nomes quando vir 127.0.0.1 (localhost) em seu arquivo */etc/resolv.conf* ou 127.0.0.1 aparecer como servidor se você executar `nslookup -debug host`.

Pode ser difícil localizar a sua configuração se você estiver executando um daemon de caching de servidor de nomes. Por padrão, o `dnsmasq` tem o arquivo de configuração */etc/dnsmasq.conf*, porém sua distribuição poderá sobrepor-lo. Por exemplo, no Ubuntu, se você configurar manualmente uma interface que tenha sido configurada pelo NetworkManager, você a encontrará no arquivo apropriado em */etc/NetworkManager/system-connections*, pois quando o NetworkManager ativa uma conexão, ele também inicia o `dnsmasq` com essa configuração. (Tudo isso pode ser sobreposto se você remover o comentário da parte referente a `dnsmasq` em seu *NetworkManager.conf*.)

O outro problema com a configuração tradicional do servidor de nomes está no fato de que ela pode ser particularmente inflexível se você quiser ser capaz de pesquisar nomes em sua rede local, sem precisar lidar com várias configurações de rede. Por exemplo, se você instalar um dispositivo de rede, vai querer ser capaz de chamá-lo pelo nome imediatamente. Isso é parte da ideia por trás dos sistemas de serviços de nome sem configuração como o mDNS (Multicast DNS) e o SSDP (Simple Service Discovery Protocol, ou Protocolo simples de descoberta de serviços). Se você quiser encontrar um host pelo nome na rede local, basta efetuar o broadcast de uma solicitação pela rede; se o host estiver presente, ele responderá com o seu endereço. Esses protocolos vão além da resolução de nomes de host ao fornecerem também informações sobre os serviços disponíveis.

A implementação Linux mais amplamente usada de mDNS chama-se Avahi. Com frequência, você verá o `mdns` como uma opção de resolução no arquivo */etc/nsswitch.conf*, no qual daremos uma olhada com mais detalhes agora.

9.12.4 */etc/nsswitch.conf*

O arquivo */etc/nsswitch.conf* controla várias configurações de precedência relacionadas a nomes em seu sistema, por exemplo, informações de usuários e de senhas; entretanto discutiremos somente as configurações de DNS neste capítulo. O arquivo em seu sistema deverá ter uma linha como esta:

```
hosts:      files dns
```

Colocar `files` na frente de `dns`, nesse caso, garante que o seu sistema verificará o arquivo */etc/hosts* para o nome de host relacionado ao endereço IP solicitado, antes de

perguntar ao servidor DNS. Normalmente, essa é uma boa ideia (em especial se você estiver pesquisando o localhost, conforme discutido a seguir), porém seu arquivo */etc/hosts* deve ser o mais *sintético* possível. Não coloque nada ali para melhorar o desempenho; fazer isso trará problemas no futuro. Você pode colocar todos os hosts de uma pequena LAN privada em */etc/hosts*, porém a regra geral é que se um host em particular tiver uma entrada DNS, ele não terá lugar em */etc/hosts*. (O arquivo */etc/hosts* também é útil para resolver nomes de host nos primeiros estágios de booting, quando a rede poderá não estar disponível ainda.)

👍 **Observação:** o DNS é um assunto amplo. Se você tiver qualquer responsabilidade sobre nomes de domínio, leia o livro *DNS and BIND*, 5ª edição, de Cricket Liu e Paul Albitz (O'Reilly, 2006).

9.13 Localhost

Ao executar `ifconfig`, você notará a interface *lo*:

```
lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING MTU:16436 Metric:1
```

A interface *lo* é uma interface de rede virtual chamada *loopback* porque ela faz um loop de volta para si mesma. O efeito disso é que, se você se conectar a 127.0.0.1, estará se conectando ao computador que estiver usando no momento. Quando dados de saída para localhost alcançarem a interface de rede do kernel para *lo*, o kernel simplesmente empacotará novamente esses dados como dados de entrada e os enviará de volta por meio de *lo*.

A interface de loopback *lo* geralmente é o único lugar em que você verá uma configuração de rede estática nos scripts executados durante o boot. Por exemplo, o comando `ifup` do Ubuntu lê */etc/network/interfaces*, e o Fedora usa */etc/sysconfig/network.interfaces/ifcfg-lo*. Geralmente, você poderá encontrar a configuração do dispositivo de loopback ao explorar */etc* usando `grep`.

9.14 A camada de transporte: TCP, UDP e serviços

Até agora, vimos somente como os pacotes se movem de um host para outro na Internet – em outras palavras, o *onde* da pergunta do início do capítulo. Vamos agora começar a responder à pergunta relacionada a *o quê*. É importante saber como o seu computador apresenta os dados dos pacotes recebidos de outros hosts aos seus processos em execução. É difícil e inconveniente para os programas do espaço de usuário lidar com

vários pacotes de dados brutos da maneira que o kernel faz. A flexibilidade é particularmente importante: mais de uma aplicação deve ser capaz de conversar com a rede ao mesmo tempo (por exemplo, você poderá ter email e vários outros clientes web executando).

Os protocolos da *camada de transporte* fazem a ligação entre os pacotes com dados brutos da camada de Internet e as necessidades mais sofisticadas das aplicações. Os dois protocolos de transporte mais populares são o TCP (Transmission Control Protocol, ou Protocolo de controle de transmissão) e o UDP (User Datagram Protocol). Iremos nos concentrar no TCP porque, de longe, é o protocolo mais comum em uso, porém daremos uma olhada rapidamente também no UDP.

9.14.1 Portas e conexões TCP

O TCP possibilita ter várias aplicações de rede em um computador por meio do uso de *portas* de rede. Uma porta é somente um número. Se um endereço IP é como um endereço de correspondência para um prédio, uma porta é como o número do apartamento – é uma subdivisão adicional.

Ao usar o TCP, uma aplicação abre uma *conexão* (não confunda isso com as conexões do NetworkManager) entre uma porta de seu próprio computador e uma porta de um host remoto. Por exemplo, uma aplicação como um navegador web pode abrir uma conexão entre a porta 36406 de seu próprio computador e a porta 80 de um host remoto. Do ponto de vista da aplicação, a porta 36406 é a porta local, e a porta 80 é a porta remota.

Podemos identificar uma conexão usando o par de endereços IP e os números de porta. Para ver as conexões que estão abertas no momento em seu computador, utilize `netstat`. A seguir, apresentamos um exemplo que mostra as conexões TCP: a opção `-n` desabilita a resolução de nomes de host (DNS) e `-t` limita a saída para TCP.

```
$ netstat -nt
```

```
Active Internet connections (w/o servers)
```

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	10.23.2.4:47626	10.194.79.125:5222	ESTABLISHED
tcp	0	0	10.23.2.4:41475	172.19.52.144:6667	ESTABLISHED
tcp	0	0	10.23.2.4:57132	192.168.231.135:22	ESTABLISHED

Os campos Local Address (Endereço local) e Foreign Address (Endereço remoto) mostram as conexões do ponto de vista de seu computador, portanto o computador, nesse caso, tem uma interface configurada em 10.23.2.4, e as portas 47626, 41475 e 57132 do lado local estão todas conectadas. A primeira conexão nesse exemplo mostra

a porta 47626 conectada à porta 5222 de 10.194.79.125.

9.14.2 Estabelecendo conexões TCP

Para estabelecer uma conexão na camada de transporte, um processo em um host inicia a conexão a partir de uma de suas portas locais com uma porta em um segundo host, enviando uma série especial de pacotes. Para reconhecer a conexão de entrada e responder, o segundo host deve ter um processo que fique *ouvindo* a porta correta. Normalmente, o processo que faz a conexão é chamado de *cliente*, e o que ouve (listener) é chamado de *servidor* (mais sobre esse assunto no capítulo 10).

O aspecto importante a saber sobre as portas é que o cliente escolhe uma porta em seu lado que não esteja em uso no momento, porém quase sempre se conecta a uma porta bem conhecida do lado do servidor. Lembre-se da saída a seguir do comando `netstat` da seção anterior:

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	10.23.2.4:47626	10.194.79.125:5222	ESTABLISHED

Com um pouco de ajuda, você pode ver que essa conexão provavelmente foi iniciada por um cliente local a um servidor remoto, pois a porta do lado local (47626) parece ser um número atribuído dinamicamente, enquanto a porta remota (5222) é um serviço bem conhecido (o Jabber, ou serviço de mensagem XMPP, para sermos mais específicos).

👉 **Observação:** uma porta atribuída dinamicamente chama-se porta efêmera.

Entretanto, se a porta local na saída for bem conhecida, é sinal de que um host remoto provavelmente iniciou a conexão. Nesse exemplo, o host remoto 172.24.54.234 conectou-se à porta 80 (a porta web default) no host local.

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	10.23.2.4:80	172.24.54.234:43035	ESTABLISHED

Um host remoto que se conectar ao seu computador em uma porta bem conhecida implica que um servidor em seu computador local está ouvindo essa porta. Para confirmar isso, liste todas as portas TCP que seu computador estiver ouvindo usando o comando `netstat`:

\$ netstat -ntl

Active Internet connections (only servers)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	0.0.0.0:80	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:53	0.0.0.0:*	LISTEN

--trecho omitido--

A linha com 0.0.0.0:80 como endereço local mostra que o computador local está ouvindo a porta 80, à espera de conexões de qualquer computador remoto. (Um servidor pode restringir o acesso a determinadas interfaces, como mostrado na última linha, em que algo está ouvindo à espera de conexões somente na interface localhost.) Para obter mais informações, use `lsof` para identificar o processo específico que está ouvindo (conforme será discutido na seção 10.5.1).

9.14.3 Números de porta e `/etc/services`

Como podemos saber se uma porta é bem conhecida? Não há uma única maneira de dizer, porém um bom local para começar é dando uma olhada em `/etc/services`, que traduz números de portas bem conhecidas para nomes. Esse arquivo tem formato texto simples. Você deverá ver entradas como estas:

```
ssh      22/tcp      # SSH Remote Login Protocol
smtp     25/tcp
domain   53/udp
```

A primeira coluna corresponde ao nome e a segunda indica o número da porta e o protocolo específico da camada de transporte (que pode ser diferente de TCP).

👉 **Observação:** além de `/etc/services`, um registro online de portas em <http://www.iana.org/> é controlado pelo documento de padrões de rede RFC6335.

No Linux, somente processos executando como superusuário podem usar as portas de 1 a 1023. Todos os processos de usuário podem ouvir e criar conexões a partir da porta 1024.

9.14.4 Características do TCP

O TCP é popular como protocolo da camada de transporte porque exige relativamente pouco do lado da aplicação. Um processo associado a uma aplicação precisa saber somente como abrir (ou ouvir), ler de ou escrever em uma conexão e fechá-la. Para a aplicação, é como se houvesse streams de entrada e de saída de dados; o processo é tão simples quanto trabalhar com um arquivo.

Contudo há muito trabalho ocorrendo nos bastidores. Somente para citar um, a implementação TCP deve saber como separar o stream de dados de saída de um processo em pacotes. Entretanto a parte difícil é saber como converter uma série de pacotes de entrada em um stream de dados de entrada para os processos lerem, especialmente quando os pacotes de entrada não necessariamente chegam na ordem correta. Além do mais, um host usando TCP deve efetuar verificação de erros: os pacotes podem se perder ou ser corrompidos quando enviados pela Internet, e uma

implementação TCP deve detectar e corrigir essas situações. A figura 9.3 mostra uma simplificação do modo como um host pode usar o TCP para enviar uma mensagem.

Felizmente, você não precisa saber quase nada sobre essa confusão além do fato de que a implementação TCP do Linux está principalmente no kernel e que os utilitários que trabalham com a camada de transporte tendem a manipular as estruturas de dados do kernel. Um exemplo é o sistema de filtragem de pacotes das Tabelas IP, discutido na seção 9.21.

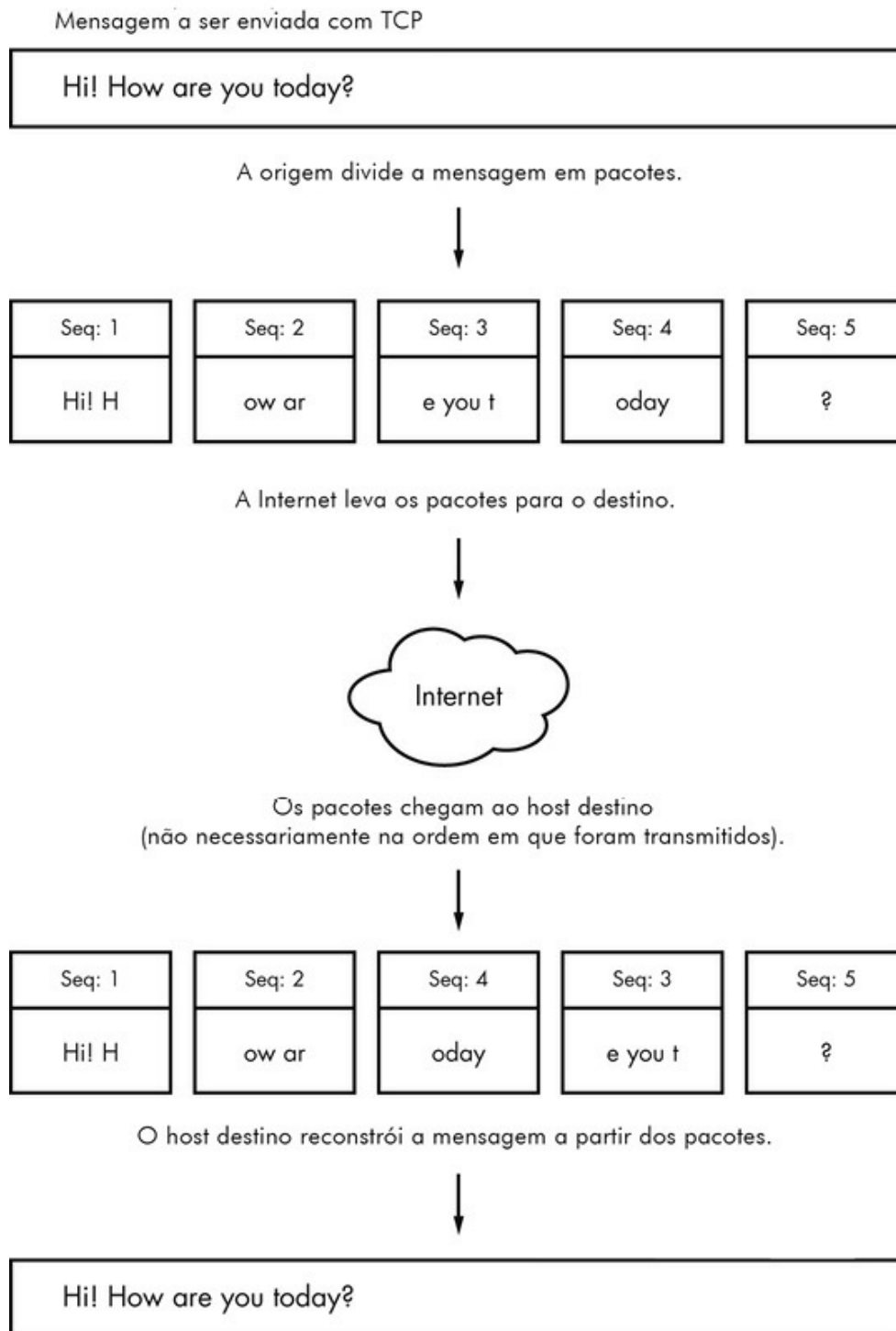


Figura 9.3 – Enviando uma mensagem com TCP.

9.14.5 UDP

O UDP é uma camada de transporte bem mais simples que o TCP. Ele define um transporte somente para mensagens únicas; não há stream de dados. Ao mesmo tempo, de modo diferente do TCP, o UDP não corrige pacotes perdidos ou fora de ordem. Com efeito, embora o UDP tenha portas, ele não tem nem mesmo conexões! Um host simplesmente envia uma mensagem a partir de uma de suas portas para uma porta em um servidor, e esse envia algo de volta, se quiser. No entanto o UDP *tem* detecção de erros para dados dentro de um pacote; um host pode detectar se um pacote foi corrompido, porém ele não precisa fazer nada a esse respeito.

Enquanto o TCP é como conversar por telefone, o UDP é como enviar uma carta, um telegrama ou uma mensagem instantânea (exceto pelo fato de que as mensagens instantâneas são mais confiáveis). As aplicações que usam UDP geralmente estão preocupadas com velocidade – enviar uma mensagem o mais rápido possível. Elas não querem ter o overhead do TCP, pois supõem que a rede entre os dois hosts geralmente é confiável. Elas não precisam da correção de erros do TCP porque têm seus próprios sistemas de detecção de erros ou porque simplesmente não se importam com eles.

Um exemplo de uma aplicação que usa UDP é o *NTP* (Network Time Protocol). Um cliente envia uma solicitação breve e simples a um servidor para obter o horário corrente, e a resposta do servidor é igualmente breve. Como o cliente quer a resposta o mais rápido possível, o UDP é adequado à aplicação; se a resposta do servidor se perder em algum ponto da rede, o cliente simplesmente enviará uma solicitação novamente ou desistirá. Outro exemplo é o bate-papo com vídeo – nesse caso, as imagens são enviadas por meio de UDP – e, se alguma parte dos dados se perder no caminho, o cliente que as estiver recebendo fará a compensação do melhor modo que puder.

 **Observação:** o restante deste capítulo trata de assuntos mais avançados sobre rede, por exemplo, filtragem na rede e roteadores, pois eles se relacionam às camadas mais baixas de rede que já vimos: as camadas física, de rede e de transporte. Se quiser, sinta-se à vontade para pular para o próximo capítulo e conhecer a camada de aplicação, em que tudo é reunido no espaço de usuário. Você verá os processos que realmente *usam* a rede em vez de simplesmente jogarem vários endereços e pacotes por aí.

9.15 Analisando uma rede local simples novamente

Agora iremos dar uma olhada nos componentes adicionais da rede simples apresentada na seção 9.3. Lembre-se de que essa rede é composta de uma rede local como sub-rede

e um roteador que conecta a sub-rede ao restante da Internet. Você aprenderá o seguinte:

- como um host da sub-rede obtém automaticamente a sua configuração de rede;
- como configurar o roteamento;
- o que é realmente um roteador;
- como saber quais endereços IP devem ser usados na sub-rede;
- como configurar firewalls para filtrar tráfego indesejado da Internet.

Vamos começar aprendendo como um host da sub-rede obtém automaticamente a sua configuração de rede.

9.16 Entendendo o DHCP

Quando um host de rede for configurado para obter sua configuração automaticamente da rede, você estará lhe dizendo para usar o DHCP (Dynamic Host Configuration Protocol, ou Protocolo de configuração dinâmica de hosts) para obter um endereço IP, a máscara de sub-rede, o gateway default e os servidores DNS. Além de fazer com que não seja necessário fornecer esses parâmetros manualmente, o DHCP tem outras vantagens para um administrador de rede, como evitar conflitos entre endereços IP e minimizar o impacto de mudanças na rede. É muito raro ver uma rede moderna que não use DHCP.

Para um host obter sua configuração com o DHCP, ele deve ser capaz de enviar mensagens a um servidor DHCP em sua rede conectada. Desse modo, cada rede física deve ter seu próprio servidor DHCP e, em uma rede simples (como aquela da seção 9.3), o roteador normalmente atua como servidor DHCP.

 **Observação:** ao fazer uma solicitação inicial ao DHCP, um host não sabe nem mesmo o endereço de um servidor DHCP, portanto ele fará um broadcast da solicitação para todos os hosts (normalmente, todos os hosts de sua rede física).

Quando um computador pede um endereço IP a um servidor DHCP, ele realmente está pedindo um *lease* (concessão) de um endereço IP por um determinado período de tempo. Quando o lease se esgotar, o cliente poderá pedir uma renovação.

9.16.1 O cliente DHCP do Linux

Embora haja vários tipos diferentes de sistemas de gerenciamento de redes, quase todos usam o programa `dhclient` do ISC (Internet Software Consortium) para fazer o verdadeiro trabalho. O `dhclient` pode ser testado manualmente na linha de comando, porém, antes de fazer isso, você *deve* remover qualquer rota de gateway default. Para executar o teste,

basta especificar o nome da interface de rede (nesse caso, é *eth0*):

```
# dhclient eth0
```

Na inicialização, o *dhclient* armazena seu ID de processo em */var/run/dhclient.pid* e suas informações de lease em */var/state/dhclient.leases*.

9.16.2 Servidores DHCP do Linux

Podemos fazer um computador Linux executar um servidor DHCP, o que provê uma boa dose de controle sobre os endereços a serem disponibilizados. Entretanto, a menos que você esteja administrando uma rede de grande porte, com várias sub-redes, provavelmente será melhor usar um hardware especializado de roteador que inclua servidores DHCP prontos.

O mais importante a saber sobre servidores DHCP é que você deverá ter apenas um executando na mesma sub-rede para evitar problemas com conflitos de endereços IP ou com configurações incorretas.

9.17 Configurando o Linux como roteador

Os roteadores, essencialmente, são apenas computadores com mais de uma interface de rede física. Você pode configurar facilmente um computador Linux como roteador.

Por exemplo, suponha que você tenha duas sub-redes LAN, 10.23.2.0/24 e 192.168.45.0/24. Para conectá-las, você tem um computador Linux que é um roteador, com três interfaces de rede: duas para as sub-redes LAN e uma para o uplink de Internet, como mostrado na figura 9.4. Conforme você pode notar, isso não parece muito diferente do exemplo de rede simples que usamos no restante deste capítulo.

Os endereços IP do roteador para as sub-redes LAN são 10.23.2.1 e 192.168.45.1. Quando esses endereços estiverem configurados, a tabela de roteamento terá uma aparência como a que se segue (os nomes da interface poderão variar na prática; ignore o uplink de Internet por enquanto):

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
10.23.2.0	0.0.0.0	255.255.255.0	U	0	0	0	eth0
192.168.45.0	0.0.0.0	255.255.255.0	U	0	0	0	eth1

Agora vamos supor que os hosts em cada sub-rede tenham o roteador como seu gateway default (10.23.2.1 para 10.23.2.0/24 e 192.168.45.1 para 192.168.45.0/24). Se 10.23.2.4 quiser enviar um pacote para qualquer host fora de 10.23.2.0/24, ele passará o pacote para 10.23.2.1. Por exemplo, para enviar um pacote de 10.23.2.4 (Host A) para 192.168.45.61 (Host E), o pacote será enviado para 10.23.2.1 (o roteador) por

meio de sua interface *eth0* e sairá pela interface *eth1* do roteador.

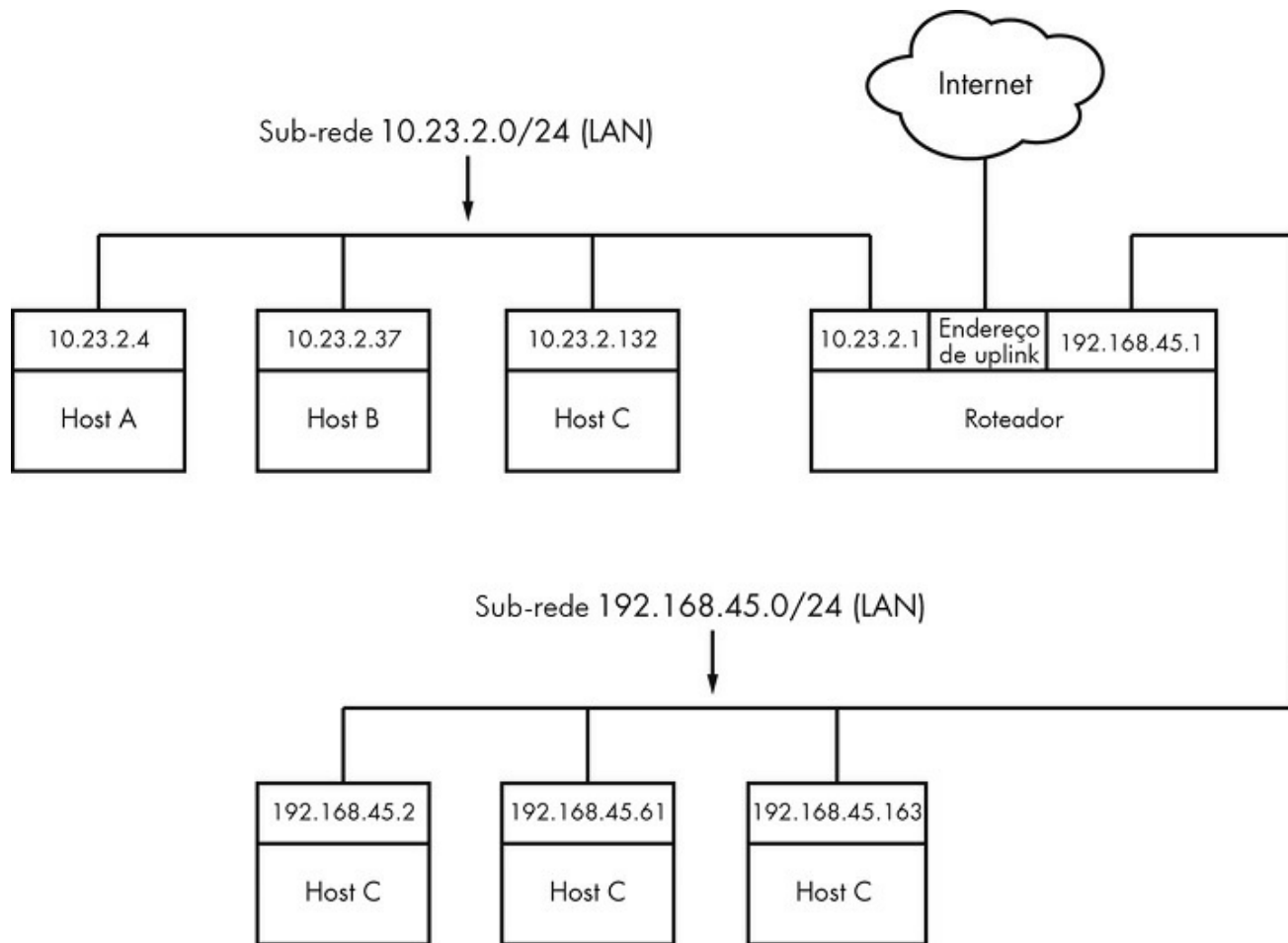


Figura 9.4 – Duas sub-redes ligadas com um roteador.

No entanto, por padrão, o kernel do Linux não transfere pacotes automaticamente de uma sub-rede para outra. Para habilitar essa função básica de roteamento, é preciso habilitar o *encaminhamento de IP* (IP forwarding) no kernel do roteador usando o comando a seguir:

```
# sysctl -w net.ipv4.ip_forward
```

Assim que você executar esse comando, o computador deverá iniciar o encaminhamento de pacotes entre as duas sub-redes, supondo que os hosts nessas sub-redes saibam enviar seus pacotes para o roteador que você acabou de criar.

Para que essa alteração seja permanente na reinicialização do sistema, você pode adicioná-la ao seu arquivo */etc/sysctl.conf*. De acordo com a sua distribuição, você poderá ter a opção de colocá-la em um arquivo em */etc/sysctl.d* para que atualizações de sua distribuição não sobrescrevam suas alterações.

9.17.1 Uplinks de Internet

Quando o roteador também tiver a terceira interface de rede com um uplink de Internet, essa mesma configuração permitirá que todos os hosts em ambas as sub-redes tenham acesso à Internet, pois estarão configurados para usar o roteador como gateway default. Porém é aí que a situação começa a ficar mais complicada. O problema é que determinados endereços IP como 10.23.2.4 não são realmente visíveis a toda a Internet; eles correspondem às chamadas *redes privadas*. Para prover conectividade com a Internet, você deve configurar um recurso chamado *NAT* (Network Address Translation, ou Tradução de endereços de rede) no roteador. O software em quase todos os roteadores especializados faz isso, portanto não há nada de extraordinário nesse caso, porém vamos analisar o problema das redes privadas mais minuciosamente.

9.18 Redes privadas

Suponha que você tenha decidido criar sua própria rede. Você tem seus computadores, o roteador e o hardware de rede prontos. Considerando o que você já sabe sobre uma rede simples até agora, sua próxima pergunta será “qual sub-rede IP devo usar?”

Se você quiser ter um bloco de endereços de Internet que todos os hosts da Internet possam ver, será possível comprar um de seu ISP. Entretanto, como a faixa de endereços IPv4 é muito limitada, isso tem um custo alto e não é muito útil além de servir para executar um servidor que o restante da Internet possa ver. A maioria das pessoas não precisa realmente desse tipo de serviço, pois elas acessam a Internet como clientes.

A alternativa convencional, de baixo custo, consiste em escolher uma sub-rede privada entre os endereços que estão nos documentos RFC 1918/6761 de padrões da Internet, mostrados na tabela 9.2.

Tabela 9.2 – Redes privadas definidas pelas RFCs 1918 e 6761

Rede	Máscara de sub-rede	Formato CIDR
10.0.0.0	255.0.0.0	10.0.0.0/8
192.168.0.0	255.255.0.0	192.168.0.0/16
172.16.0.0	255.240.0.0	172.16.0.0/12

Você pode compor as sub-redes privadas da forma que quiser. A menos que você planeje ter mais de 254 hosts em uma única rede, escolha uma sub-rede pequena, como 10.23.2.0/24, como estivemos usando ao longo deste capítulo. (As redes com essa máscara de rede às vezes são chamadas de sub-redes *classe C*. Embora, de certo modo,

seja tecnicamente obsoleto, o termo continua sendo útil.)

Qual é o truque? Os hosts na Internet de verdade não sabem nada sobre as sub-redes privadas e não enviarão pacotes a elas, portanto, sem alguma ajuda, os hosts das sub-redes privadas não poderão conversar com o mundo externo. Um roteador conectado à Internet (com um endereço verdadeiro, que não seja privado) deve ter alguma maneira de preencher a lacuna entre essa conexão e os hosts em uma rede privada.

9.19 Tradução de endereços de rede (Mascaramento de IP)

O NAT é a maneira mais comumente usada de compartilhar um único endereço IP com uma rede privada, e é de uso quase universal em redes domésticas e de pequenos escritórios. No Linux, a variante do NAT usada pela maioria das pessoas é conhecida como *mascaramento de IP* (IP masquerading).

A ideia básica por trás do NAT é que o roteador não só transferirá pacotes de uma sub-rede para outra, mas também os transformará antes de transferi-los. Os hosts da Internet sabem como se conectar ao roteador, porém não sabem nada sobre a rede privada por trás dele. Os hosts da rede privada não precisam de nenhuma configuração especial; o roteador será seu gateway default.

O sistema, de modo geral, funciona da seguinte maneira:

1. Um host da rede privada interna quer fazer uma conexão com o mundo externo, portanto ele enviará seus pacotes de solicitação de conexão por meio do roteador.
2. O roteador intercepta o pacote de solicitação de conexão em vez de passá-lo para a Internet (em que ele se perderia porque a Internet pública não sabe nada sobre as redes privadas).
3. O roteador determina o destino do pacote de solicitação de conexão e abre sua própria conexão com o destino.
4. Quando o roteador obtém a conexão, ele simula uma mensagem de “conexão estabelecida” para o host interno original.
5. O roteador agora é o intermediário entre o host interno e o destino. O destino não sabe nada sobre o host interno; para o host remoto, a conexão parecerá ter vindo do roteador.

Isso não é tão simples quanto parece. O roteamento IP normal conhece apenas os endereços IP de origem e de destino na camada de Internet. Entretanto, se o roteador

lidasse somente com a camada de Internet, cada host da rede interna poderia estabelecer somente uma conexão com um único destino de cada vez (entre outras limitações), pois não há nenhuma informação na parte referente à camada de Internet de um pacote para distinguir várias solicitações do mesmo host para o mesmo destino. Sendo assim, o NAT deve ir além da camada de Internet e dissecar os pacotes para extrair mais informações de identificação, particularmente os números de porta UDP e TCP das camadas de transporte. O UDP é razoavelmente simples porque há portas, mas não há conexões, porém a camada de transporte TCP é mais complexa.

Para configurar um computador Linux para que atue como um roteador NAT, você deve ativar tudo o que está a seguir na configuração do kernel: filtragem de pacotes da rede (“suporte a firewall”), monitoração de conexões, suporte a tabelas IP, NAT completo e suporte a alvo MASQUERADE. A maioria dos kernels das distribuições já vem com esse suporte.

Em seguida, será preciso executar alguns comandos `iptables` de aparência complexa para fazer o roteador executar NAT para sua sub-rede privada. A seguir, apresentamos um exemplo que se aplica a uma rede Ethernet interna em *eth1* compartilhando uma conexão externa em *eth0* (você aprenderá mais sobre a sintaxe do `iptables` na seção 9.21).

```
# sysctl -w net.ipv4.ip_forward
# iptables -P FORWARD DROP
# iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
# iptables -A FORWARD -i eth0 -o eth1 -m state --state ESTABLISHED,RELATED -j ACCEPT
# iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT
```

 **Observação:** embora o NAT funcione bem na prática, lembre-se de que ele é essencialmente um hack usado para estender o tempo de vida do espaço de endereços IPv4. Em um mundo perfeito, todos estariam utilizando o IPv6 (a Internet de próxima geração) e usariam seu espaço de endereçamento maior e mais sofisticado sem que houvesse qualquer dificuldade.

É provável que você jamais vá precisar usar os comandos anteriores, a menos que esteja desenvolvendo seu próprio software, especialmente com tantos hardwares de roteadores com propósitos especiais disponíveis. Porém a função do Linux em uma rede não termina aqui.

9.20 Roteadores e o Linux

Nos primórdios da era da banda larga, os usuários com menos exigência quanto à demanda simplesmente conectavam seus computadores diretamente à Internet. Porém não demorou muito para que muitos usuários quisessem compartilhar uma única conexão de banda larga com suas próprias redes, e os usuários de Linux em particular,

com frequência, configuravam um computador extra para ser usado como roteador executando NAT.

Os fabricantes responderam a esse novo mercado oferecendo hardwares de roteador especializados, constituídos de um processador eficiente, memória flash e várias portas de rede – com capacidade suficiente para administrar uma rede simples comum, executar softwares importantes como um servidor DHCP e usar o NAT. No que diz respeito ao software, muitos fabricantes lançaram mão do Linux para capacitar seus roteadores. Eles adicionaram os recursos de kernel necessários, reduziram os softwares do espaço de usuário e criaram interfaces de administração baseadas em GUI.

Quase ao mesmo tempo em que esses primeiros roteadores apareceram, muitas pessoas se interessaram em explorar mais o hardware. Foi solicitado que um fabricante – o Linksys – disponibilizasse o código-fonte de seu software sob os termos da licença de um de seus componentes, e logo, distribuições especializadas de Linux como o OpenWRT surgiram para os roteadores. (O “WRT” nesses nomes é proveniente do número de modelo do Linksys.)

Além do aspecto relativo ao hobby, há bons motivos para usar essas distribuições: geralmente, elas são mais estáveis que o firmware dos fabricantes, especialmente em hardwares mais antigos de roteadores, e elas normalmente oferecem recursos adicionais. Por exemplo, para conectar uma rede por meio de uma conexão wireless, muitos fabricantes exigem que você compre um hardware correspondente, porém, com o OpenWRT instalado, o fabricante e a idade do hardware realmente não importam. Isso ocorre porque você estará usando um sistema operacional realmente aberto no roteador, que não se importa com o hardware que você use, desde que ele seja suportado.

Você pode utilizar boa parte do conhecimento presente neste livro para analisar o funcionamento interno de firmwares personalizados de Linux, embora você vá encontrar diferenças, especialmente ao fazer login. Como ocorre com muitos sistemas embarcados, os firmwares abertos tendem a usar o BusyBox para oferecer vários recursos de shell. O BusyBox é um programa executável único que oferece funcionalidades limitadas para vários comandos Unix como o shell, ls, grep, cat e more. (Isso faz com que uma quantidade significativa de memória seja economizada.) Além do mais, o init usado no momento de boot tende a ser bem simples nos sistemas embarcados. Entretanto, normalmente, você não verá essas limitações como um problema, pois um firmware personalizado de Linux, em geral, inclui uma interface web para administração, semelhante ao que seria disponibilizado por um fabricante.

9.21 Firewalls

Os roteadores, em particular, devem sempre incluir algum tipo de firewall para manter o tráfego indesejado longe de sua rede. Um *firewall* é uma configuração de software e/ou de hardware que normalmente está presente em um roteador, entre a Internet e uma rede menor, tentando garantir que nada “ruim” da Internet cause danos à rede menor. Podemos também configurar recursos de firewall para cada computador; nesse caso, o computador verificará todos os dados de entrada e de saída no nível de pacotes (em vez de isso ser feito na camada de aplicação, em que os programas servidores normalmente tentam realizar algum tipo de controle de acesso próprio). O processo de configurar o firewall em computadores individuais às vezes é chamado de *filtragem IP* (IP filtering).

Um sistema pode filtrar pacotes quando:

- receber um pacote;
- enviar um pacote;
- encaminhar (rotear) um pacote para outro host ou gateway.

Sem firewalls instalados, um sistema simplesmente processará os pacotes e os enviará para o seu destino. Os firewalls colocam pontos de verificação (checkpoints) para os pacotes nos pontos de transferência de dados identificados anteriormente. Os pontos de verificação descartam, rejeitam ou aceitam pacotes, normalmente de acordo com alguns desses critérios:

- o endereço IP de origem ou de destino ou a sub-rede;
- a porta de origem ou de destino (nas informações da camada de transporte);
- a interface de rede do firewall.

Os firewalls oferecem uma oportunidade para trabalhar com o subsistema do kernel do Linux que processa pacotes IP. Vamos discutir isso a seguir.

9.21.1 Básico sobre firewalls no Linux

No Linux, as regras de firewall são criadas em uma série conhecida como *cadeia* (chain). Um conjunto de cadeias forma uma *tabela*. À medida que um pacote se move pelas diversas partes do subsistema de rede do Linux, o kernel aplica as regras de determinadas cadeias aos pacotes. Por exemplo, após receber um novo pacote da camada física, o kernel ativa as regras de cadeias correspondentes à entrada.

Todas essas estruturas de dados são mantidas pelo kernel. O sistema como um todo é

chamado de *iptables*, e há um comando `iptables` do espaço de usuário para criar e manipular as regras.

👍 **Observação:** há um sistema mais novo chamado *nftables* cujo objetivo é substituir o *iptables*, porém, na época desta publicação, o *iptables* ainda era o sistema dominante para firewalls.

Pelo fato de poder haver diversas tabelas – cada qual com seus próprios conjuntos de cadeias, cada um contendo várias regras –, o fluxo de pacotes pode se tornar bem complicado. Contudo, normalmente, você trabalhará principalmente com uma única tabela chamada *filter* (filtro) que controla o fluxo básico dos pacotes. Há três cadeias básicas na tabela *filter*: INPUT para pacotes de entrada, OUTPUT para pacotes de saída e FORWARD para pacotes encaminhados.

As figuras 9.5 e 9.6 mostram diagramas de fluxo simplificados que exibem os locais em que as regras são aplicadas aos pacotes na tabela *filter*. Há duas figuras porque os pacotes podem entrar no sistema a partir de uma interface de rede (Figura 9.5) ou podem ser gerados por um processo local (Figura 9.6). Como você pode ver, um pacote de entrada na rede pode ser consumido por um processo de usuário e pode não alcançar a cadeia FORWARD ou a cadeia OUTPUT. Os pacotes gerados pelos processos de usuário não alcançarão as cadeias INPUT ou FORWARD.

Isso se torna mais complicado porque há vários passos ao longo do caminho, além de somente essas três cadeias. Por exemplo, os pacotes estão sujeitos às cadeias PREROUTING e POSTROUTING, e o processamento das cadeias também poderá ocorrer em qualquer um dos três níveis mais baixos de rede. Para ver um diagrama grande com tudo o que está ocorrendo, pesquise na Internet em busca de “Linux netfilter packet flow” (fluxo de pacotes netfilter do Linux), porém lembre-se de que esses diagramas tentam incluir todo cenário possível para a entrada e o fluxo de pacotes. Geralmente, separar os diagramas de acordo com a origem do pacote ajuda, como vimos nas figuras 9.5 e 9.6.

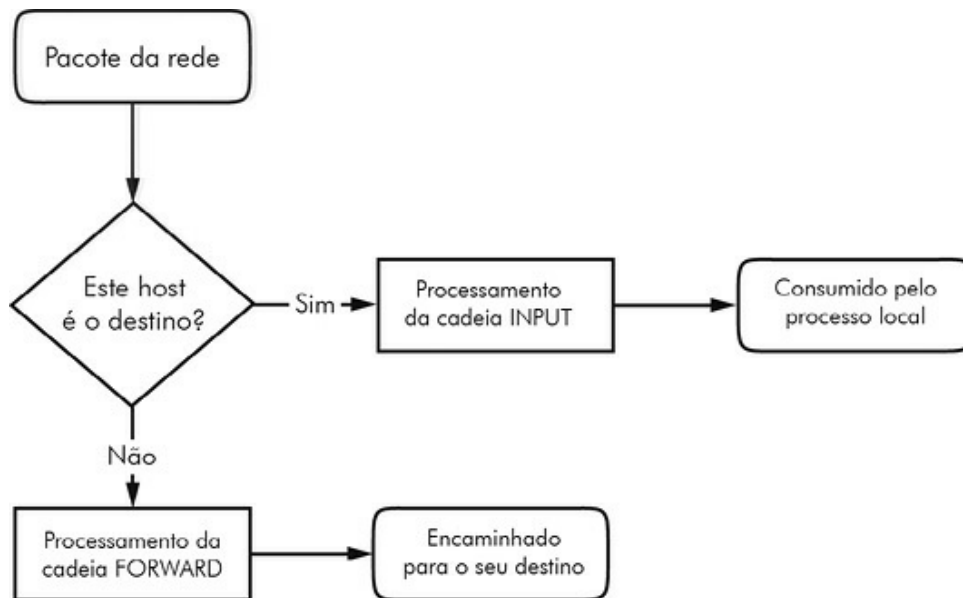


Figura 9.5 – Sequência de processamento de cadeias para pacotes de entrada de uma rede.



Figura 9.6 – Sequência de processamento de cadeias para pacotes de entrada de um processo local.

9.21.2 Configurando as regras do firewall

Vamos ver como o sistema de tabelas IP funciona na prática. Comece visualizando a configuração atual com o comando a seguir:

```
# iptables -L
```

A saída normalmente será um conjunto de cadeias vazio, como o que se segue:

```
Chain INPUT (policy ACCEPT)
target prot opt source destination
Chain FORWARD (policy ACCEPT)
Target prot opt source destination
Chain OUTPUT (policy ACCEPT)
target prot opt source destination
```

Cada cadeia do firewall tem uma *política* (policy) default que especifica o que deve ser feito com um pacote se não houver nenhuma regra que corresponda ao pacote. A política para todas as três cadeias nesse exemplo é ACCEPT, que significa que o kernel permitirá que o pacote passe pelo sistema de filtragem de pacotes. A política DROP diz ao kernel para descartar o pacote. Para definir a política em uma cadeia, utilize `iptables -P` da seguinte maneira:

iptables -P FORWARD DROP

👍 **Aviso:** não faça nada imprudente com as políticas em seu computador até ter lido o restante desta seção.

Suponha que alguém em 192.168.34.63 esteja importunando você. Para evitar que esse host converse com o seu computador, execute o comando a seguir:

iptables -A INPUT -s 192.168.34.63 -j DROP

O parâmetro `-A INPUT` concatena uma regra à cadeia INPUT. A parte referente a `-s 192.168.34.63` especifica o endereço IP de origem na regra, e `-j DROP` diz ao kernel para descartar qualquer pacote que corresponda à regra. Desse modo, seu computador jogará fora qualquer pacote proveniente de 192.168.34.63.

Para ver a regra definida, execute `iptables -L`:

```
Chain INPUT (policy ACCEPT)
Target     prot opt source      destination
DROP      all  --  192.168.34.63    anywhere
```

Infelizmente, seu amigo em 192.168.34.63 disse a todos da sub-rede dele que abrissem conexões com a sua porta SMTP (porta TCP 25). Para se livrar também desse tráfego, execute:

iptables -A INPUT -s 192.168.34.0/24 -p tcp --destination-port 25 -j DROP

Esse exemplo adiciona um qualificador de máscara de rede ao endereço de origem, bem como `-p tcp` para especificar somente pacotes TCP. Outra restrição, `--destination-port 25`, diz que a regra deve ser aplicada somente ao tráfego na porta 25. A lista da tabela IP para INPUT agora tem o seguinte aspecto:

```
Chain INPUT (policy ACCEPT)
Target     prot opt source      destination
DROP      all  --  192.168.34.63    anywhere
DROP      tcp  --  192.168.34.0/24  anywhere    tcp dpt:smtp
```

Tudo está bem até você ouvir alguém conhecido em 192.168.34.37 dizendo que não pode enviar emails a você porque você bloqueou seus computadores. Pensando que esta seja uma correção rápida, você executa o comando a seguir:

iptables -A INPUT -s 192.168.34.37 -j ACCEPT

Entretanto ela não funciona. Para saber o motivo, dê uma olhada na nova cadeia:

```
Chain INPUT (policy ACCEPT)
Target     prot opt source      destination
DROP      all  --  192.168.34.63    anywhere
DROP      tcp  --  192.168.34.0/24  anywhere    tcp dpt:smtp
ACCEPT    all  --  192.168.34.37    anywhere
```

O kernel lê a cadeia de cima para baixo, usando a primeira regra que permitir uma correspondência.

A primeira regra não faz a correspondência com 192.168.34.37, mas a segunda faz, pois ela se aplica a todos os hosts de 192.168.34.1 a 192.168.34.254, e essa segunda regra diz para descartar os pacotes. Quando houver correspondência com uma regra, o kernel executará a ação e não olhará mais para baixo na cadeia. (Você poderá perceber que 192.168.34.37 pode enviar pacotes para qualquer porta em seu computador, *exceto* para a porta 25, porque a segunda regra se aplica *somente* à porta 25.)

A solução consiste em mover a terceira regra para cima. Inicialmente, apague a terceira regra usando o comando a seguir:

```
# iptables -D INPUT 3
```

Em seguida, *insira* essa regra no topo da cadeia usando iptables -I:

```
# iptables -I INPUT -s 192.168.34.37 -j ACCEPT
```

Para inserir uma regra em outro ponto de uma cadeia, coloque o número da regra após o nome da cadeia (por exemplo, iptables -I INPUT 4 ...).

9.21.3 Estratégias de firewall

Embora o tutorial anterior tenha mostrado como inserir regras e como o kernel processa cadeias IP, ainda não vimos estratégias de firewall que realmente funcionem. Vamos falar disso agora.

Há dois tipos básicos de cenários para firewall: um para proteger computadores individuais (em que as regras são definidas na cadeia INPUT de cada computador) e um para proteger uma rede de computadores (em que as regras são definidas na cadeia FORWARD de um roteador). Em ambos os casos, você não poderá ter um nível sério de segurança se usar uma política default igual a ACCEPT e inserir regras continuamente para descartar pacotes de origens que comecem a enviar dados inapropriados. Somente os pacotes em que você confiar devem ser permitidos, e tudo o mais deverá ser proibido.

Por exemplo, suponha que o seu computador tenha um servidor SSH na porta TCP 22. Não há motivo algum para um host qualquer iniciar uma conexão com qualquer outra porta de seu computador, e você não deve dar essa chance a nenhum host. Para configurar isso, inicialmente defina a política da cadeia INPUT com DROP:

```
# iptables -P INPUT DROP
```

Para permitir tráfego ICMP (para ping e outros utilitários), utilize a linha a seguir:

```
# iptables -A INPUT -p icmp -j ACCEPT
```

Certifique-se de que você poderá receber pacotes que você enviar tanto para o seu próprio endereço IP quanto para 127.0.0.1 (localhost). Supondo que o endereço IP de seu host seja *meu_ender*, faça o seguinte:

```
# iptables -A INPUT -s 127.0.0.1 -j ACCEPT
```

```
# iptables -A INPUT -s meu_ender -j ACCEPT
```

Se você controlar toda a sua sub-rede (e confiar totalmente nela), poderá substituir *meu_ender* pelo seu endereço de sub-rede e pela máscara de sub-rede, por exemplo, 10.23.2.0/24.

Porém, embora você continue querendo impedir conexões TCP de entrada, será preciso garantir que seu host possa fazer conexões TCP com o mundo externo. Como todas as conexões TCP começam com um pacote SYN (solicitação de conexão), se você permitir que todos os pacotes TCP que não sejam pacotes SYN passem, não haverá problemas:

```
# iptables -A INPUT -p tcp '!' --syn -j ACCEPT
```

Em seguida, se um DNS remoto baseado em UDP estiver sendo usado, você deverá aceitar o tráfego de seu servidor de nomes para que o seu computador possa pesquisar nomes junto ao DNS. Faça isso para *todos* os servidores DNS em */etc/resolv.conf*. Utilize o comando a seguir (em que o endereço do servidor de nomes é *ender_sn*):

```
# iptables -A INPUT -p udp --source-port 53 -s ender_sn -j ACCEPT
```

Por fim, permita conexões SSH de qualquer lugar:

```
# iptables -A INPUT -p tcp --destination-port 22 -j ACCEPT
```

As configurações anteriores de iptables funcionam em diversas situações, incluindo qualquer conexão direta (especialmente de banda larga), em que é muito mais provável que um invasor faça um scan das portas de seu computador. Você também pode adaptar essas configurações para um roteador com firewall usando a cadeia FORWARD no lugar de INPUT e usando sub-redes de origem e de destino nos locais em que forem apropriadas. Para observar configurações mais avançadas, uma ferramenta de configuração como o Shorewall poderá ser útil.

Essa discussão mal tocou a superfície sobre políticas de segurança. Lembre-se de que a ideia principal é permitir somente aquilo que você achar aceitável, em vez de tentar identificar e eliminar o que for ruim. Além do mais, o firewall para IPs é somente uma parte do quadro geral referente à segurança. (Você verá mais sobre esse assunto no próximo capítulo.)

9.22 Ethernet, IP e ARP

Há um detalhe básico interessante na implementação do IP sobre Ethernet que ainda não discutimos. Lembre-se de que um host deve colocar um pacote IP dentro de um frame Ethernet para transmitir o pacote pela camada física para outro host. Lembre-se também de que os próprios frames não incluem informações sobre endereço IP; eles usam endereços MAC (de hardware). A pergunta é: ao criar o frame Ethernet para um pacote IP, como o host sabe qual endereço MAC corresponde ao endereço IP de destino?

Normalmente, não pensamos nessa pergunta, principalmente porque o software de rede inclui um sistema automático de pesquisa de endereços MAC chamado *ARP* (Address Resolution Protocol, ou Protocolo de resolução de endereços). Um host que use Ethernet como sua camada física e IP como a camada de rede mantém uma pequena tabela chamada *ARP cache* que mapeia endereços IP a endereços MAC. No Linux, a ARP cache está no kernel. Para visualizar a ARP cache de seu computador, utilize o comando `arp`. (Como ocorre com vários outros comandos de rede, a opção `-n`, nesse caso, desabilita pesquisas reversas de DNS.)

```
$ arp -n
```

Address	Hwtype	Hwaddr	Flags	Mask	Iface
10.1.2.141	ether	00:11:32:0d:ca:82	C		eth0
10.1.2.1	ether	00:24:a5:b5:a0:11	C		eth0
10.1.2.50	ether	00:0c:41:f6:1c:99	C		eth0

Quando um computador é inicializado, sua ARP cache está vazia. Então como esses endereços MAC vão para a cache? Tudo começa quando o computador quer enviar um pacote para outro host. Se um endereço IP alvo não estiver em uma ARP cache, os passos a seguir serão executados:

1. O host origem criará um frame Ethernet especial contendo um pacote de solicitação ARP para o endereço MAC, que corresponda ao endereço IP alvo.
2. O host origem fará o broadcast desse frame para toda a rede física da sub-rede do alvo.
3. Se um dos outros hosts da sub-rede conhecer o endereço MAC correto, ele criará um pacote de resposta e um frame contendo o endereço, e o enviará de volta à origem. Com frequência, o host que responder *será* o host-alvo e estará simplesmente respondendo com o seu próprio endereço MAC.
4. O host origem adicionará o par de endereços IP-MAC à ARP cache e poderá prosseguir.

👍 **Observação:** lembre-se de que o ARP aplica-se somente aos computadores em sub-redes locais

(consulte a seção 9.4 para ver suas sub-redes locais). Para alcançar destinos fora de sua sub-rede, seu host enviará o pacote para o roteador e, depois disso, o problema não será mais dele. É claro que seu host ainda precisa saber o endereço MAC do roteador, e ele pode usar o ARP para descobri-lo.

O único verdadeiro problema que você pode ter com o ARP é que a cache de seu sistema poderá ficar desatualizada se você mudar um endereço IP de uma placa de interface de rede para outra, pois as placas têm endereços MAC diferentes (por exemplo, ao testar um computador). Os sistemas Unix invalidam as entradas da ARP cache se não houver nenhuma atividade após um período de tempo, portanto não deverá haver nenhum problema além de um pequeno atraso no caso de dados invalidados, porém você pode apagar uma entrada da ARP cache imediatamente usando o comando a seguir:

```
# arp -d host
```

Também podemos ver a ARP cache para uma única interface de rede usando:

```
$ arp -i interface
```

A página de manual `arp(8)` explica como definir entradas da ARP cache manualmente, porém você não deverá precisar fazer isso.

 **Observação:** não confunda ARP com RARP (Reverse Address Resolution Protocol, ou Protocolo de resolução reversa de endereços). O RARP transforma um endereço MAC de volta para um nome de host ou um endereço IP. Antes de o DHCP ter se tornado popular, algumas estações de trabalho sem disco e outros dispositivos usavam RARP para obter suas configurações, porém o RARP é raro nos dias de hoje.

9.23 Ethernet wireless

Em princípio, as redes Ethernet wireless (“WiFi”) não são muito diferentes das redes com fio. De modo muito parecido com qualquer hardware com fio, elas têm endereços MAC e usam frames Ethernet para transmitir e receber dados; como resultado, o kernel do Linux pode conversar com uma interface de rede wireless de modo muito semelhante ao que faria com uma interface de rede com fio. Tudo na camada de rede e acima dela é igual; as principais diferenças são componentes adicionais na camada física como frequências, IDs de rede, segurança e assim por diante.

De modo diferente do hardware de rede com fio, que é muito bom em se ajustar automaticamente às nuances da instalação física sem muita complicação, a configuração das redes wireless é muito mais aberta. Para fazer uma interface wireless funcionar adequadamente, o Linux precisa de ferramentas adicionais de configuração.

Vamos dar uma olhada rápida nos componentes adicionais das redes wireless.

- Detalhes de transmissão – são as características físicas, por exemplo, a

frequência de rádio.

- Identificação da rede – como mais de uma rede wireless pode compartilhar o mesmo meio físico básico, você deverá ser capaz de fazer a distinção entre elas. O SSID (Service Set Identifier, também conhecido como “nome da rede”) é o identificador da rede wireless.
- Gerenciamento – embora seja possível configurar redes wireless para fazer os hosts conversarem diretamente uns com os outros, a maioria das redes wireless é administrada por um ou mais *pontos de acesso* (access points) pelos quais todo o tráfego passa. Os pontos de acesso geralmente ligam uma rede wireless a uma rede com fio, fazendo com que ambas pareçam ser uma única rede.
- Autenticação – você pode querer restringir o acesso a uma rede wireless. Para isso, é possível configurar pontos de acesso que exijam uma senha ou outra chave de autenticação antes que elas possam sequer conversar com um cliente.
- Criptografia – além de restringir o acesso inicial a uma rede wireless, normalmente você vai querer criptografar todo o tráfego que for enviado por ondas de rádio.

A configuração do Linux e os utilitários que lidam com esses componentes estão espalhados por diversas áreas. Alguns estão no kernel: o Linux tem um conjunto de extensões para wireless que padroniza o acesso do espaço de usuário ao hardware. No que concerne ao espaço de usuário, a configuração para wireless pode se tornar complicada, portanto a maioria das pessoas prefere usar frontends GUI, como o applet de desktop para NetworkManager, para fazer com que tudo funcione. Mesmo assim, vale a pena dar uma olhada em algumas das atividades que ocorrem nos bastidores.

9.23.1 iw

Você pode visualizar e alterar dispositivos do espaço do kernel e configurações de rede usando um utilitário chamado *iw*. Para usá-lo, normalmente será preciso saber o nome da interface de rede para o dispositivo, por exemplo, *wlan0*. A seguir, apresentamos um exemplo que mostra um scan de redes wireless disponíveis. (Espere muitos dados de saída se você estiver em uma área urbana.)

```
# iw dev wlan0 scan
```

 **Observação:** a interface de rede deverá estar ativa para que esse comando funcione (se não estiver, execute `ifconfig wlan0 up`), porém não será necessário configurar nenhum parâmetro da camada de rede, por exemplo, um endereço IP.

Se a interface de rede estiver conectada a uma rede wireless, você poderá ver os

detalhes da rede da seguinte maneira:

```
# iw dev wlan0 link
```

O endereço MAC na saída desse comando é do ponto de acesso com o qual você está conversando no momento.

👉 **Observação:** o comando `iw` distingue entre os nomes de dispositivos físicos, como *phy0*, e os nomes de interface de rede, como *wlan0*, e permite que você altere várias configurações para cada um deles. É possível até mesmo criar mais de uma interface de rede para um único dispositivo físico. Entretanto, em quase todos os casos básicos, você simplesmente usará o nome da interface de rede.

Utilize `iw` para conectar uma interface de rede a uma rede wireless que não seja segura do seguinte modo:

```
# iw wlan0 connect nome_da_rede
```

Conectar-se a redes seguras é outra história. Para o sistema WEP (Wired Equivalent Privacy) sem segurança, você poderá usar o parâmetro `keys` com o comando `iw connect`. No entanto você não deverá usar WEP se estiver levando a segurança a sério.

9.23.2 Segurança em redes wireless

Na maioria das instalações wireless seguras, o Linux conta com um daemon chamado `wpa_supplicant` para gerenciar tanto a autenticação quanto a criptografia para uma interface de rede wireless. Esse daemon pode cuidar tanto de esquemas de autenticação WPA (WiFi Protected Access, ou Acesso WiFi protegido) quanto WPA2, assim como de quase todo tipo de técnica de criptografia usado em redes wireless. Quando o daemon é iniciado pela primeira vez, ele lê um arquivo de configuração (por padrão, é `/etc/wpa_supplicant.conf`) e tenta se identificar junto a um ponto de acesso e estabelecer comunicação de acordo com um nome de rede especificado. O sistema está bem documentado; em particular, as páginas de manual `wpa_supplicant(1)` e `wpa_supplicant.conf(5)` estão bem detalhadas.

Executar o daemon manualmente sempre que você quiser estabelecer uma conexão dá bastante trabalho. Com efeito, somente criar o arquivo de configuração é uma tarefa enfadonha em virtude da quantidade de opções possíveis. Para piorar mais ainda a situação, todo o trabalho de executar `iw` e `wpa_supplicant` simplesmente permite que seu sistema se conecte a uma rede wireless física; a camada de rede não é sequer configurada. É nesse ponto que gerenciadores automáticos de configuração de rede como o NetworkManager eliminam muitas das dificuldades do processo. Embora não façam nenhum trabalho sozinhos, eles conhecem a sequência correta e a configuração necessária para cada passo em direção a ter uma rede wireless operacional.

9.24 Resumo

Você pode agora perceber que entender as posições e as funções das várias camadas de rede é fundamental para compreender como as redes no Linux funcionam e saber como efetuar a configuração das redes. Embora tenhamos discutido somente o básico, assuntos mais avançados relacionados às camadas física, de rede e de transporte apresentam semelhanças com o que vimos. As próprias camadas frequentemente são subdivididas, como você acabou de ver com as diversas partes da camada física em uma rede wireless.

Uma quantidade substancial de ação que vimos neste capítulo ocorre no kernel, com alguns utilitários de controle básicos no espaço de usuário para manipular as estruturas de dados internas do kernel (por exemplo, as tabelas de roteamento). Essa é a maneira tradicional de trabalhar com redes. Entretanto, como ocorre com vários dos assuntos discutidos neste livro, algumas tarefas não são adequadas ao kernel graças à complexidade e à necessidade de ter flexibilidade, e é aí que os utilitários do espaço de usuário assumem o controle. Em particular, o NetworkManager monitora e consulta o kernel e então manipula a sua configuração. Outro exemplo é o suporte a protocolos dinâmicos de roteamento como o BGP (Border Gateway Protocol), que é usado em roteadores de Internet de grande porte.

Porém é provável que a essa altura você esteja um pouco entediado com configurações de rede. Vamos nos concentrar no *uso* da rede – a camada de aplicação.

CAPÍTULO 10

Aplicações e serviços de rede

Este capítulo explora as aplicações básicas de rede – clientes e servidores que executam no espaço de usuário e estão na camada de aplicação. Como essa camada está na parte superior da pilha, próxima aos usuários finais, você poderá achar esse material mais acessível que o capítulo 9. Realmente, você interage todos os dias com aplicações de rede clientes, como navegadores web e programas para leitura de emails.

Para fazerem seu trabalho, os clientes de rede se conectam aos servidores de rede correspondentes. Os servidores de rede Unix se apresentam em diversas formas. Um programa servidor pode ouvir uma porta por conta própria ou por meio de um servidor secundário. Além do mais, os servidores não têm nenhum banco de dados de configuração comum, porém têm uma ampla variedade de funcionalidades. A maioria dos servidores tem um arquivo de configuração para controlar seu comportamento (embora não tenham um formato comum), e a maior parte deles usa o serviço syslog do sistema operacional para fazer logging de mensagens. Daremos uma olhada em alguns servidores comuns bem como em algumas ferramentas que ajudarão você a entender e a depurar a operação dos servidores.

Os clientes de rede usam os protocolos da camada de transporte e as interfaces do sistema operacional, portanto entender o básico sobre as camadas de transporte TCP e UDP é importante. Vamos começar dando uma olhada nas aplicações de rede ao usar um cliente de rede que utiliza TCP.

10.1 Básico sobre serviços

Os serviços TCP estão entre os mais fáceis de entender porque estão baseados em streams de dados bidirecionais simples e ininterruptos. Talvez a melhor maneira de ver como eles funcionam seja conversar diretamente com um servidor web na porta TCP 80 para ter uma ideia de como os dados fluem pela conexão. Por exemplo, execute o comando a seguir para se conectar a um servidor web:

```
$ telnet www.wikipedia.org 80
```

Você deverá obter uma resposta como esta:

Trying *algum endereço*...
Connected to www.wikipedia.org.
Escape character is '^['.

Agora digite:

GET / HTTP/1.0

Tecle Enter duas vezes. O servidor deverá enviar uma porção de texto HTML como resposta e, em seguida, encerrará a conexão.

Esse exercício nos diz que:

- o host remoto tem um processo relacionado a um servidor web ouvindo a porta TCP 80;
- o telnet foi o cliente que iniciou a conexão.

 **Observação:** o telnet é um programa criado originalmente para permitir logins em hosts remotos. Embora o servidor de login remoto telnet não baseado em Kerberos seja totalmente desprovido de segurança (como você verá posteriormente), o cliente telnet pode ser útil para depurar serviços remotos. O telnet não trabalha com UDP nem com qualquer camada de transporte que não seja o TCP. Se você estiver procurando um cliente de rede de propósito geral, considere o netcat, que será descrito na seção 10.5.3.

10.1.1 Um olhar mais detalhado

No exemplo anterior, interagimos manualmente com um servidor web na rede com o telnet, usando o protocolo de camada de aplicação HTTP (Hypertext Transfer Protocol, ou Protocolo de transferência de hipertexto). Embora normalmente um navegador web seja usado para fazer esse tipo de conexão, vamos dar um passo além do telnet e usar um programa de linha de comando que saiba conversar com a camada de aplicação HTTP. Usaremos o utilitário `curl` com uma opção especial para registrar os detalhes dessa comunicação:

```
$ curl --trace-ascii arquivo_de_trace http://www.wikipedia.org/
```

 **Observação:** sua distribuição pode não ter o pacote `curl` previamente instalado, porém você não deverá ter problemas para instalá-lo, se for necessário.

Você obterá muitos dados HTML na saída. Ignore-os (ou redirecione-os para `/dev/null`) e observe o arquivo *arquivo_de_trace* recém-criado. Supondo que a conexão tenha sido bem-sucedida, a primeira parte do arquivo deverá ter um aspecto como o que se segue, no ponto em que `curl` tenta estabelecer a conexão TCP com o servidor:

```
== Info: About to connect() to www.wikipedia.org port 80 (#0)  
== Info: Trying 10.80.154.224... == Info: connected
```

Tudo que você viu até agora ocorre na camada de transporte ou abaixo dela. Entretanto, se essa conexão for bem-sucedida, `curl` tentará enviar a solicitação (o “header”, ou

cabeçalho); é nesse ponto que a camada de aplicação começa:

```
=> Send header, 167 bytes (0xa7)
0000: GET / HTTP/1.1
0010: User-Agent: curl/7.22.0 (i686-pc-linux-gnu) libcurl/7.22.0 OpenS
0050: SL/1.0.1 zlib/1.2.3.4 libidn/1.23 librtmp/2.3
007f: Host: www.wikipedia.org
0098: Accept: */*
00a5:
```

A primeira linha, nesse caso, é a saída de debugging do `curl` informando o que ele fará em seguida. As linhas restantes mostram o que o `curl` envia para o servidor. O texto em negrito corresponde ao que é enviado ao servidor; os números hexadecimais no início são somente offsets para debugging do `curl` para ajudar você a monitorar a quantidade de dados enviada ou recebida.

Você pode ver que o `curl` começa enviando um comando `GET` ao servidor (como você fez com o `telnet`), seguido de algumas informações extras para ele e uma linha vazia. Em seguida, o servidor envia uma resposta, inicialmente com seu próprio cabeçalho, mostrado em negrito a seguir:

```
<= Recv header, 17 bytes (0x11)
0000: HTTP/1.1 200 OK
<= Recv header, 16 bytes (0x10)
0000: Server: Apache
<= Recv header, 42 bytes (0x2a)
0000: X-Powered-By: PHP/5.3.10-1ubuntu3.9+wmfl
--trecho omitido--
```

De modo muito semelhante à saída anterior, as linhas `<=` representam saídas de debugging, e `0000:` antecede as linhas de saída para informar os offsets.

O cabeçalho na resposta do servidor pode ser bem longo, porém, em algum ponto, o servidor fará a transição de envio de cabeçalhos para o envio do documento solicitado, da seguinte maneira:

```
<= Recv header, 55 bytes (0x37)
0000: X-Cache: cp1055 hit (16), cp1054 frontend hit (22384)
<= Recv header, 2 bytes (0x2)
0000:
<= Recv data, 877 bytes (0x36d)
0000: 008000
0008: <!DOCTYPE html>.<html lang="mul" dir="ltr">.<head>.<!-- Sysops:
--trecho omitido--
```

Essa saída também mostra uma propriedade importante da camada de aplicação.

Embora a saída de debugging informe `Recv header` e `Recv data`, indicando que há dois tipos diferentes de mensagem do servidor, não há nenhuma diferença no modo como o `curl` conversou com o sistema operacional para obter os dois tipos de mensagem, nem qualquer diferença na maneira como o sistema operacional os tratou ou como a rede tratou os pacotes subjacentes. A diferença está totalmente na própria aplicação `curl` do espaço de usuário. O `curl` sabia que, até esse ponto, ele estava obtendo os cabeçalhos, porém, quando recebeu uma linha em branco (a parte com 2 bytes no meio) indicando o final dos cabeçalhos no HTTP, ele soube interpretar tudo o que se seguiu como sendo o documento solicitado.

O mesmo é válido para o servidor que enviou esses dados. Ao enviar a resposta, o servidor não fez nenhuma diferenciação entre o cabeçalho e os dados do documento enviados ao sistema operacional; as distinções ocorrem dentro do programa servidor no espaço de usuário.

10.2 Servidores de rede

Em sua maioria, os servidores de rede são como outros daemons servidores de seu sistema – como, por exemplo, o `cron` –, exceto pelo fato de eles interagirem com portas de rede. Com efeito, lembre-se do `syslogd` discutido no capítulo 7; ele aceita pacotes UDP na porta 514 quando é iniciado com a opção `-r`.

A seguir, apresentamos outros servidores de rede comuns que você poderá encontrar executando em seu sistema:

- `httpd`, `apache`, `apache2` – servidores web;
- `sshd` – daemon secure shell (veja a seção 10.3);
- `postfix`, `qmail`, `sendmail` – servidores de emails;
- `cupsd` – servidor de impressão;
- `nfsd`, `mountd` – daemons de sistemas de arquivos para rede (compartilhamento de arquivos);
- `smbd`, `nmdbd` – daemons de compartilhamento de arquivos para Windows (veja o capítulo 12);
- `rpcbind` – daemon de serviço de mapeamento de portas RPC (Remote Procedure Call, ou Chamada de procedimento remoto).

Um recurso comum à maioria dos servidores de rede é que eles geralmente funcionam como vários processos. Pelo menos um processo ouve uma porta de rede, e quando uma nova conexão de entrada é recebida, o processo que estiver ouvindo utiliza `fork()` para

criar um novo processo-filho que, por sua vez, é responsável pela nova conexão. O processo-filho, geralmente chamado de processo *worker*, termina quando a conexão é encerrada. Enquanto isso, o processo que estava ouvindo originalmente continua a ouvir a porta da rede. Esse processo permite que um servidor cuide facilmente de várias conexões sem que haja muitos problemas.

Porém há algumas exceções a esse modelo. Chamar `fork()` adiciona uma quantidade significativa de overhead ao sistema. Em comparação, os servidores TCP de alto desempenho como o servidor web Apache podem criar vários processos worker na inicialização para que esses já estejam prontos para lidar com as conexões à medida que for necessário. Os servidores que aceitam pacotes UDP simplesmente recebem dados e reagem a eles; esses servidores não precisam monitorar conexões.

10.3 Secure Shell (SSH)

Todo servidor funciona de um modo um pouco diferente. Vamos dar uma olhada mais de perto em um deles: o servidor SSH autônomo. Uma das aplicações de serviço de rede mais comuns é o SSH (secure shell), que é padrão de mercado para acesso remoto a um computador Unix. Quando configurado, o SSH permite logins seguros em shell, execução remota de programas, compartilhamento simples de arquivos e outros recursos – substituindo os antigos sistemas de acesso remoto `telnet` e `rlogin` por criptografia de chave pública para autenticação e cifradores mais simples para dados de sessão. A maioria dos ISPs e provedores na nuvem exigem SSH para acesso de shell a seus serviços, e muitos dispositivos de rede baseados em Linux (como os dispositivos NAS) também permitem acesso por meio de SSH. O OpenSSH (<http://www.openssh.com/>) é uma implementação popular gratuita de SSH para Unix, e quase todas as distribuições Linux vêm com ele previamente instalado. O cliente OpenSSH é o `ssh`, e o servidor é `sshd`. Há duas versões principais do protocolo SSH: 1 e 2. O OpenSSH suporta ambas, porém a versão 1 raramente é usada.

Entre as várias funcionalidades e os recursos úteis, o SSH faz o seguinte:

- Criptografa sua senha e todos os demais dados de sessão, protegendo você dos bisbilhoteiros.
- Faz o tunelamento de outras conexões de rede, incluindo aquelas de clientes X Window System. Você aprenderá mais sobre o X no capítulo 14.
- Oferece clientes para quase todos os sistemas operacionais.
- Usa chaves para autenticação de hosts.

👍 **Observação:** tunelamento (tunneling) é o processo de empacotar e transportar uma conexão de rede usando outra. As vantagens de usar o SSH para fazer o tunelamento de conexões X Window System estão no fato de o SSH configurar o ambiente de display para você e de criptografar os dados X do túnel.

O SSH, porém, tem suas desvantagens. Uma delas é que, para estabelecer uma conexão SSH, é preciso ter a chave pública do host remoto, e você não necessariamente a obterá de maneira segura (embora possa verificá-la manualmente para garantir que não está sendo enganado). Para ter uma visão geral de como os diversos métodos de criptografia funcionam, adquira o livro *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2ª edição, de Bruce Schneier (Wiley, 1996). Dois livros detalhados sobre o SSH são: *SSH Mastery: OpenSSH, PuTTY, Tunnels and Keys* de Michael W. Lucas (Tilted Windmill Press, 2012) e *SSH, The Secure Shell*, 2ª edição, de Daniel J. Barrett, Richard E. Silverman e Robert G. Byrnes (O'Reilly, 2005).

10.3.1 O servidor SSHD

Executar o `sshd` exige um arquivo de configuração e chaves de host. A maioria das distribuições mantém as configurações no diretório de configuração `/etc/ssh` e tenta configurar tudo apropriadamente se você instalar seus pacotes `sshd`. (O nome do arquivo de configuração `sshd_config` é fácil de ser confundido com o arquivo de configuração `ssh_config` do cliente, portanto tome cuidado.)

Você não deverá precisar mudar nada em `sshd_config`, mas dar uma olhada nele não fará mal a ninguém. O arquivo é constituído de pares palavra-chave/valor, conforme mostrado no fragmento a seguir:

```
Port 22
#Protocol 2,1
#ListenAddress 0.0.0.0
#ListenAddress ::
HostKey /etc/ssh/ssh_host_key
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key
```

As linhas começadas com `#` são comentários, e vários comentários em seu `sshd_config` podem indicar valores default. A página de manual `sshd_config(5)` contém descrições de todos os valores possíveis, porém os mais importantes são:

- `HostKey arquivo` — usa `arquivo` como chave de um host. (As chaves de host serão descritas em breve.)
- `LogLevel nível` — faz log de mensagens com nível de `syslog nível`.
- `PermitRootLogin valor` — permite ao superusuário fazer login no SSH se `valor` estiver

definido com *yes*. Defina *valor* com *no* para evitar isso.

- `SyslogFacility nome` — faz log de mensagens com a facilidade *nome* no syslog.
- `X11Forwarding valor` — habilita o tunelamento de clientes X Window System se *valor* estiver definido com *yes*.
- `XAuthLocation path` — fornece um path para *xauth*. O tunelamento de X11 não funcionará sem esse path. Se *xauth* não estiver em */usr/bin*, defina *path* com o nome completo do path para *xauth*.

Chaves de host

O OpenSSH tem três conjuntos de chaves de host: um para o protocolo versão 1 e dois para o protocolo 2. Cada conjunto tem uma *chave pública* (com uma extensão de arquivo *.pub*) e uma *chave privada* (sem extensão). Não deixe ninguém ver sua chave privada, mesmo em seu próprio sistema, pois, se alguém acessá-la, você correrá o risco de sofrer invasões.

O SSH versão 1 tem somente chaves RSA, enquanto o SSH versão 2 tem chaves RSA e DSA. RSA e DSA são algoritmos de criptografia de chave pública. Os nomes de arquivo das chaves podem ser vistos na tabela 10.1.

Normalmente, não será necessário criar as chaves porque o programa de instalação do OpenSSH ou o script de instalação de sua distribuição farão isso por você, mas é preciso saber como criá-las se você planeja usar programas como o *ssh-agent*. Para criar chaves para a versão 2 do protocolo SSH, utilize o programa *ssh-keygen* que vem com o OpenSSH:

```
# ssh-keygen -t rsa -N "" -f /etc/ssh/ssh_host_rsa_key
# ssh-keygen -t dsa -N "" -f /etc/ssh/ssh_host_dsa_key
```

Tabela 10.1 – Arquivos de chaves do OpenSSH

Nome do arquivo	Tipo de chave
<i>ssh_host_rsa_key</i>	Chave RSA privada (versão 2)
<i>ssh_host_rsa_key.pub</i>	Chave RSA pública (versão 2)
<i>ssh_host_dsa_key</i>	Chave DSA privada (versão 2)
<i>ssh_host_dsa_key.pub</i>	Chave DSA pública (versão 2)
<i>ssh_host_key</i>	Chave RSA privada (versão 1)
<i>ssh_host_key.pub</i>	Chave RSA pública (versão 1)

Para criar chaves para a versão 1, utilize:

```
# ssh-keygen -t rsa1 -N "" -f /etc/ssh/ssh_host_key
```

O servidor e os clientes SSH também usam um arquivo de chave chamado *ssh_known_hosts*, que contém chaves públicas de outros hosts. Se você pretende usar autenticação baseada em hosts, o arquivo *ssh_known_hosts* do servidor deverá conter as chaves públicas dos hosts para todos os clientes confiáveis. Saber a respeito dos arquivos de chave será útil se você estiver substituindo um computador. Ao instalar um novo computador do zero, você poderá importar os arquivos de chaves do computador antigo para garantir que não haverá discrepância nas chaves quando os usuários se conectarem ao novo computador.

Iniciando o servidor SSH

Embora a maioria das distribuições venha com o SSH, o servidor *sshd* não é iniciado por padrão. No Ubuntu e no Debian, instalar o pacote do servidor SSH faz com que as chaves sejam criadas, inicia o servidor e adiciona a inicialização à configuração de boot. No Fedora, o *sshd* é instalado por padrão, mas não é ativado. Para iniciar o *sshd* no boot, utilize *chkconfig* da seguinte maneira (esse comando não iniciará o serviço imediatamente; utilize *service sshd start* para isso):

```
# chkconfig sshd on
```

O Fedora normalmente cria qualquer arquivo de chaves de host que estiver faltando na primeira inicialização do *sshd*.

Se você não tiver nenhum suporte do *init* instalado ainda, executar o *sshd* como *root* iniciará o servidor e, na inicialização, o *sshd* escreverá seu PID em */var/run/sshd.pid*. O *sshd* também pode ser iniciado como uma unidade de socket no *systemd* ou com *inetd*, porém, normalmente, fazer isso não é uma boa ideia, pois, ocasionalmente, o servidor precisará gerar os arquivos de chave – um processo que pode demorar bastante.

10.3.2 O cliente SSH

Para fazer login em um host remoto, execute:

```
$ ssh nome_de_usuario_remoto@host
```

nome_de_usuario_remoto@ pode ser omitido se seu nome de usuário local for o mesmo nome em *host*. Você também pode executar pipelines de e para um comando *ssh*, como mostrado no exemplo a seguir, que copia um diretório *dir* para outro host:

```
$ tar zcvf - dir | ssh host_remoto tar zxvf -
```

O arquivo de configuração global do cliente SSH – *ssh_config* – deve estar em

/etc/ssh, juntamente com seu arquivo *sshd_config*. Como ocorre com o arquivo de configuração do servidor, o arquivo de configuração do cliente tem pares chave-valor, porém você não precisará alterá-los.

O problema mais frequente ao usar clientes SSH ocorre quando uma chave pública SSH em seu arquivo *ssh_known_hosts* ou *.ssh/known_hosts* local não coincide com a chave no host remoto. Chaves ruins causam erros ou avisos como este:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED! @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
38:c2:f6:0d:0d:49:d4:05:55:68:54:2a:2f:83:06:11.
Please contact your system administrator.
Add correct host key in /home/user/.ssh/known_hosts to get rid of this message.
Offending key in /home/user/.ssh/known_hosts:12❶
RSA host key for host has changed and you have requested
strict checking.
Host key verification failed.
```

Geralmente, isso simplesmente quer dizer que o administrador do host remoto alterou as chaves (com frequência, isso ocorre quando o hardware é substituído), porém não faz mal algum dar uma verificada junto ao administrador, se você não tiver certeza. Qualquer que seja o caso, a mensagem anterior diz que a chave ruim está na linha 12 do arquivo *known_hosts* de um usuário, como mostrado em ❶.

Se você não suspeita de uma situação desonesta, basta remover a linha ofensora ou substituí-la pela chave pública correta.

Clientes SSH para transferência de arquivos

O OpenSSH inclui os programas *scp* e *sftp* para transferência de arquivos, cujo propósito é servir de substitutos para os programas *rcp* e *ftp* mais antigos e desprovidos de segurança. O *scp* pode ser usado para transferir arquivos de um computador remoto para o seu computador ou vice-versa, ou de um host para outro. Ele funciona como o comando *cp*. A seguir, apresentamos alguns exemplos:

```
$ scp usuário@host:arquivo .
$ scp arquivo usuário@host:dir
$ scp usuário1@host1:arquivo usuário2@host2:dir
```

O programa *sftp* funciona como o cliente de linha de comando *ftp* e usa os comandos *get* e

put. O host remoto deve ter um programa `sftp-server` instalado, que você poderá esperar que esteja presente se o host remoto também usar OpenSSH.

 **Observação:** se precisar de mais recursos e flexibilidade além dos oferecidos por `scp` e `sftp` (por exemplo, se você transferir um grande número de arquivos com frequência), dê uma olhada em `rsync`, descrito no capítulo 12.

Clientes SSH para plataformas diferentes de Unix

Há clientes SSH para todos os sistemas operacionais populares, conforme listado na página web do OpenSSH (<http://www.openssh.com/>). Qual deles você deve escolher? O PuTTY é um bom cliente Windows básico que inclui um programa seguro para cópia de arquivos. O MacSSH funciona bem no Mac OS 9.x e em versões anteriores. O Mac OS X é baseado no Unix e inclui o OpenSSH.

10.4 Os daemons `inetd` e `xinetd`

Implementar servidores autônomos para todos os serviços, de certo modo, pode ser ineficiente. Cada servidor deve ser configurado separadamente para lidar com a tarefa de ouvir portas, fazer controle de acesso e efetuar configuração de portas. Essas ações são realizadas da mesma maneira para a maioria dos serviços; somente quando um servidor aceitar uma conexão é que haverá uma diferença no modo como essa será tratada.

Uma maneira tradicional de simplificar o uso dos servidores é usando o daemon `inetd` — um tipo de *superservidor* criado para padronizar o acesso às portas de rede e as interfaces entre programas servidores e portas de rede. Após iniciar o `inetd`, ele lerá seu arquivo de configuração e ficará ouvindo as portas de rede definidas nesse arquivo. À medida que novas conexões de rede surgirem, o `inetd` associará um processo recém-criado à conexão.

Uma versão mais recente do `inetd` chamada `xinetd` permite uma configuração mais simples e oferece um controle de acesso melhor, porém o próprio `xinetd` está sendo descontinuado em favor do `systemd`, que pode prover as mesmas funcionalidades por meio de unidades de socket, conforme descrito na seção 6.4.7.

Embora o `inetd` não seja mais comumente usado, sua configuração mostra tudo que é necessário para configurar um serviço. O fato é que o `sshd` também pode ser chamado pelo `inetd` em vez de ser usado como servidor autônomo, como mostrado no arquivo de configuração `/etc/inetd.conf` a seguir:

```
ident stream      tcp  nowait  root    /usr/sbin/sshd  sshd -i
```

Nesse caso, os sete campos, da esquerda para a direita, são:

- Nome do serviço – o nome do serviço em */etc/services* (veja a seção 9.14.3).
- Tipo de socket – geralmente é *stream* para TCP e *dgram* para UDP.
- Protocolo – o protocolo de transporte; geralmente, é *tcp* ou *udp*.
- Comportamento do servidor de datagramas – para UDP, é *wait* ou *nowait*. Os serviços que se utilizam de outros protocolos de transporte devem usar *nowait*.
- Usuário – o nome do usuário que executará o serviço. Adicione *.group* para definir um grupo.
- Executável – o programa que *inetd* deve conectar ao serviço.
- Argumentos: os argumentos para o executável. O primeiro argumento deve ser o nome do programa.

10.4.1 Wrappers para TCP: *tcpd*, */etc/hosts.allow* e */etc/hosts.deny*

Antes de os firewalls de mais baixo nível terem se tornado populares, muitos administradores usavam a biblioteca TCP *wrapper* e um daemon para controle de hosts sobre serviços de rede. Nessas implementações, *inetd* executa o programa *tcpd* que, inicialmente, observa a conexão de entrada bem como as listas de controle de acesso nos arquivos */etc/hosts.allow* e */etc/hosts.deny*. O programa *tcpd* faz log da conexão e, se decidir que a conexão de entrada não apresenta problemas, ele a passará para o programa de serviço final. (Embora você ainda possa encontrar um sistema que continue usando o sistema TCP wrapper, não o discutiremos em detalhes porque ele caiu amplamente em desuso.)

10.5 Ferramentas de diagnóstico

Vamos dar uma olhada em algumas ferramentas de diagnóstico úteis para observar a camada de aplicação. Algumas exploram as camadas de transporte e de rede, pois tudo o que está na camada de aplicação em algum momento será mapeado para algo presente nessas camadas inferiores.

Conforme discutimos no capítulo 9, o *netstat* é uma ferramenta básica de debugging de serviços de rede que pode exibir várias estatísticas das camadas de transporte e de rede. A tabela 10.2 apresenta algumas opções úteis para visualizar conexões.

Tabela 10.2 – Opções úteis do netstat para informações sobre conexões

Opção	Descrição

-t	Exibe informações de porta TCP
-u	Exibe informações de porta UDP
-l	Exibe as portas que estão sendo ouvidas
-a	Exibe todas as portas ativas
-n	Desabilita a pesquisa de nomes (provê agilidade; será útil também se o DNS não estiver funcionando)

10.5.1 lsof

No capítulo 8, você aprendeu que o `lsof` pode monitorar arquivos abertos, porém ele também pode listar os programas que estiverem usando ou ouvindo portas no momento. Para obter uma lista completa dos programas que estão usando ou ouvindo portas, execute `lsof -i`.

Quando executado como um usuário normal, esse comando mostra somente os processos desse usuário. Quando executado como `root`, a saída deverá ter uma aparência semelhante a que se segue, exibindo uma variedade de processos e de usuários:

```
COMMAND  PID  USER  FD  TYPE  DEVICE  SIZE/OFF  NODE NAME
rpcbind  700  root   6u  IPv4  10492   0t0  UDP *:sunrpc
rpcbind  700  root   8u  IPv4  10508   0t0  TCP *:sunrpc (LISTEN)
avahi-daemon 872  avahi 13u  IPv4 21736375 0t0  UDP *:mdns
cupsd    1010 root   9u  IPv6 42321174 0t0  TCP ip6-localhost:ipp (LISTEN)
ssh      14366 juser   3u  IPv4 38995911 0t0  TCP thishost.local:55457->
        somehost.example.com:ssh (ESTABLISHED)
chromium- 26534 juser   8r  IPv4 42525253 0t0  TCP thishost.local:41551->
        anotherhost.example.com:https (ESTABLISHED)
```

Esse exemplo de saída mostra os usuários e os IDs de processo para programas servidores e clientes, desde serviços RPC de estilo antigo no início e um serviço DNS multicast disponibilizado pelo `avahi` até um serviço de impressão preparado para IPv6 (`cupsd`). As duas últimas entradas mostram conexões de clientes: uma conexão SSH e uma conexão de web segura do navegador web Chromium. Como a saída pode ser longa, normalmente é melhor aplicar um filtro (conforme será discutido na próxima seção).

O programa `lsof` é como `netstat` no sentido que tenta resolver de forma reversa todo endereço IP que encontrar, transformando-o em um nome de host, o que deixa a saída lenta. Utilize a opção `-n` para desabilitar a resolução de nomes:

```
# lsof -n -i
```

Você também pode especificar `-P` para desabilitar as pesquisas de nomes de porta em `/etc/services`.

Filtrando de acordo com o protocolo e a porta

Se você estiver procurando uma porta em particular (suponha que você saiba que um processo está usando uma determinada porta e queira saber qual é esse processo), utilize o comando a seguir:

```
# lsof -i:porta
```

A sintaxe completa é a seguinte:

```
# lsof -iprotocolo@host:porta
```

Os parâmetros *protocolo*, *@host* e *:porta* são todos opcionais e filtrarão a saída de `lsof` de acordo com seus valores. Como ocorre com a maioria dos utilitários de rede, *host* e *porta* podem ser nomes ou números. Por exemplo, se você quiser ver somente as conexões na porta TCP 80 (a porta HTTP), utilize:

```
# lsof -iTCP:80
```

Filtrando de acordo com o status da conexão

Um filtro particularmente prático de `lsof` é relacionado ao status da conexão. Por exemplo, para mostrar somente os processos que estão ouvindo portas TCP, digite:

```
# lsof -iTCP -sTCP:LISTEN
```

Esse comando oferece uma boa visão geral dos processos servidores de rede que estão executando em seu sistema no momento. No entanto, como os servidores UDP não ficam ouvindo portas nem têm conexões, você deverá usar `-iUDP` para ver os clientes em execução bem como os servidores. Normalmente, isso não é um problema porque provavelmente não haverá muitos servidores UDP em seu sistema.

10.5.2 tcpdump

Se você precisar ver exatamente quais dados estão passando pela sua rede, o `tcpdump` colocará sua placa de interface de rede em *modo promíscuo* e dará informações sobre todos os pacotes que passarem por ela. Digitar `tcpdump` sem argumentos irá gerar uma saída semelhante a que se segue, que inclui uma solicitação ARP e uma conexão web:

```
# tcpdump
```

```
tcpdump: listening on eth0
```

```
20:36:25.771304 arp who-has mikado.example.com tell duplex.example.com
```

```
20:36:25.774729 arp reply mikado.example.com is-at 0:2:2d:b:ee:4e
```

```
20:36:25.774796 duplex.example.com.48455 > mikado.example.com.www: S
```

```

3200063165:3200063165(0) win 5840 <mss 1460,sackOK,timestamp 38815804[[tcp]]> (DF)
20:36:25.779283 mikado.example.com.www > duplex.example.com.48455: S
3494716463:3494716463(0) ack 3200063166 win 5792 <mss 1460,sackOK,timestamp 4620[[tcp]]> (DF)
20:36:25.779409 duplex.example.com.48455 > mikado.example.com.www: . ack 1 win
5840 <nop,nop,timestamp 38815805 4620> (DF)
20:36:25.779787 duplex.example.com.48455 > mikado.example.com.www: P 1:427(426)
ack 1 win 5840 <nop,nop,timestamp 38815805 4620> (DF)
20:36:25.784012 mikado.example.com.www > duplex.example.com.48455: . ack 427
win 6432 <nop,nop,timestamp 4620 38815805> (DF)
20:36:25.845645 mikado.example.com.www > duplex.example.com.48455: P 1:773(772)
ack 427 win 6432 <nop,nop,timestamp 4626 38815805> (DF)
20:36:25.845732 duplex.example.com.48455 > mikado.example.com.www: . ack 773
win 6948 <nop,nop,timestamp 38815812 4626> (DF)

9 packets received by filter
0 packets dropped by kernel

```

Você pode dizer ao tcpdump para ser mais específico ao adicionar filtros. É possível filtrar de acordo com hosts de origem e de destino, redes, endereços Ethernet, protocolos nas várias camadas diferentes do modelo de rede e muito mais. Entre os vários protocolos de pacotes que o tcpdump reconhece estão ARP, RARP, ICMP, TCP, UDP, IP, IPv6, AppleTalk e pacotes IPX. Por exemplo, para dizer ao tcpdump para apresentar somente pacotes TCP, execute:

```
# tcpdump tcp
```

Para ver pacotes web e pacotes UDP, digite:

```
# tcpdump udp or port 80
```

 **Observação:** se você precisar fazer muito sniffing de pacotes, considere usar uma alternativa com GUI ao tcpdump, por exemplo, o Wireshark.

Primitivas

Nos exemplos anteriores, tcp, udp e port 80 são chamados de *primitivas*. As primitivas mais importantes estão na tabela 10.3:

Tabela 10.3 – Primitivas do tcpdump

Primitiva	Especificação do pacote
tcp	Pacotes TCP
udp	Pacotes UDP
port <i>porta</i>	Pacotes TCP e/ou UDP de e para a porta <i>porta</i>
host <i>host</i>	Pacotes de ou para <i>host</i>

Operadores

O `or` usado no exemplo anterior é um *operador*. O `tcpdump` pode usar vários operadores (como `and` e `!`), e os operadores podem ser agrupados entre parênteses. Se você planeja realizar algum trabalho sério com o `tcpdump`, não se esqueça de ler a página do manual, especialmente a seção que descreve as primitivas.

Quando o `tcpdump` não deve ser usado

Tome muito cuidado ao usar o `tcpdump`. A saída do `tcpdump` mostrada anteriormente nesta seção inclui somente pacotes TCP (camada de transporte) e informações de cabeçalho IP (camada de Internet), mas você também pode fazer o `tcpdump` exibir o conteúdo todo dos pacotes. Apesar de muitos operadores de rede fazerem com que a tarefa de observar pacotes de rede seja muito simples, você não deve bisbilhotar as redes, a menos que elas sejam suas.

10.5.3 netcat

Se precisar ter mais flexibilidade para se conectar a um host remoto além do que um comando como `telnet host porta` permite, utilize o `netcat` (ou `nc`). O `netcat` pode se conectar a portas TCP/UDP remotas, permitir a especificação de uma porta local, ouvir portas, fazer scan de portas, redirecionar I/O padrão de e para conexões de rede etc. Para abrir uma conexão TCP com uma porta usando `netcat`, execute:

```
$ netcat host porta
```

O `netcat` só termina quando o outro lado da conexão encerrá-la, o que pode deixar a situação confusa se você redirecionar a saída-padrão para o `netcat`. A conexão pode ser encerrada a qualquer momento com `Ctrl-C`. (Se quiser que o programa e a conexão de rede terminem com base no stream de entrada-padrão, experimente usar o programa `sock` em seu lugar.)

Para ouvir uma porta em particular, execute:

```
$ netcat -l -p número_da_porta
```

10.5.4 Scanning de portas

Às vezes, você não sabe sequer quais serviços os computadores de suas redes estão oferecendo nem mesmo quais endereços IP estão em uso. O programa `Nmap` (Network Mapper) faz scan de todas as portas em um computador ou em uma rede de computadores à procura de portas abertas e lista as portas que encontrar. A maioria das

distribuições tem um pacote Nmap, ou você pode obtê-lo em <http://www.insecure.org/>. (Veja a página de manual do Nmap e acesse recursos online para saber tudo o que o Nmap pode fazer.)

Quando estiver ouvindo portas em seu próprio computador, geralmente será útil executar o scan do Nmap a partir de pelo menos dois pontos: de seu próprio computador e de outro (possivelmente, fora de sua rede local). Fazer isso proporcionará uma visão geral do que o seu firewall está bloqueando.

👉 **AVISO:** se alguém mais controlar a rede em que você quer efetuar um scan com o Nmap, peça permissão. Os administradores de rede observam scans de portas e geralmente desabilitam o acesso aos computadores que os estiverem executando.

Execute `nmap host` para executar um scan genérico em um host. Por exemplo:

\$ **nmap 10.1.2.2**

Starting Nmap 5.21 (<http://nmap.org>) at 2015-09-21 16:51 PST

Nmap scan report for 10.1.2.2

Host is up (0.00027s latency).

Not shown: 993 closed ports

PORT STATE SERVICE

22/tcp open ssh

25/tcp open smtp

80/tcp open http

111/tcp open rpcbind

8800/tcp open unknown

9000/tcp open cslistener

9090/tcp open zeus-admin

Nmap done: 1 IP address (1 host up) scanned in 0.12 seconds

Como você pode ver, vários serviços estão abertos nesse caso, muitos dos quais não estão habilitados por padrão na maioria das distribuições. Com efeito, o único serviço aqui que está normalmente habilitado por padrão é a porta 111 – a porta `rpcbind`.

10.6 Remote Procedure Call (RPC)

O que podemos dizer do serviço `rpcbind` que acabamos de ver no scan da seção anterior? RPC quer dizer *Remote Procedure Call* (Chamada de procedimento remoto) – um sistema localizado nas partes mais baixas da camada de aplicação. Ele foi concebido para fazer com que fosse mais fácil aos programadores acessar aplicações de rede ao tirar vantagem do fato de os programas chamarem funções em programas remotos (identificados por números de programa) e os programas retornarem um código de resultado ou uma mensagem.

As implementações de RPC usam protocolos de transporte como TCP e UDP, e elas exigem um serviço intermediário especial para mapear números de programa a portas TCP e UDP. O servidor chama-se `rpcbind` e deverá executar em qualquer computador que queira usar serviços RPC.

Para ver quais serviços RPC o seu computador tem, execute:

```
$ rpcinfo -p localhost
```

O RPC é um daqueles protocolos que simplesmente não querem morrer. Os sistemas NFS (Network File System) e NIS (Network Information Service) usam RPC, porém eles são totalmente desnecessários em computadores autônomos. No entanto, sempre que você acha que eliminou toda a necessidade do `rpcbind`, algo diferente surge, por exemplo, o suporte ao FAM (File Access Monitor) no GNOME.

10.7 Segurança em redes

Pelo fato de o Linux ser uma variante bem popular do Unix na plataforma PC e, especialmente pelo fato de ser amplamente usado para servidores web, ele atrai muitas personalidades desagradáveis que tentam invadir os sistemas de computadores. A seção 9.21 discutiu os firewalls, porém essa não é toda a história quando se trata de segurança.

A segurança de redes atrai extremistas – aqueles que *realmente* gostam de invadir sistemas (seja por diversão ou por dinheiro) e aqueles que surgem com esquemas sofisticados de proteção que *realmente* gostam de repelir as pessoas que tentam invadir seus sistemas. (Isso também pode ser muito rentável.) Felizmente, não é preciso saber muito para manter seu sistema seguro. A seguir, apresentamos algumas regras gerais básicas:

- Execute o mínimo possível de serviços – os invasores não podem atacar serviços que não existam em seu sistema. Se você souber o que é um serviço e não o estiver usando, não habilite esse serviço somente pelo fato de poder querer usá-lo “em algum momento no futuro”.
- Bloqueie o máximo possível usando um firewall – os sistemas Unix têm diversos serviços internos que podem não ser de seu conhecimento (como a porta TCP 111 para o servidor de mapeamento de portas RPC), e nenhum outro sistema no mundo *deverá* saber sobre eles. Pode ser bem difícil monitorar e controlar os serviços em seu sistema, pois vários tipos diferentes de programas ouvem várias portas. Para evitar que os invasores descubram os serviços internos existentes em seu sistema, use regras eficientes de firewall e instale um em seu roteador.

- Monitore os serviços que você oferecer na Internet – se você executar um servidor SSH, Postfix ou serviços semelhantes, mantenha seu software atualizado e obtenha alertas apropriados de segurança. (Veja a seção 10.7.2 para conhecer alguns recursos online.)
- Use versões de distribuição com “suporte de longo prazo” para os servidores – as equipes de segurança normalmente concentram seus trabalhos em versões de distribuição estáveis e com suporte. As versões de desenvolvimento e de testes como o Debian Unstable e o Fedora Rawhide recebem muito menos atenção.
- Não dê uma conta de seu sistema a ninguém que não precise de uma – é muito mais fácil conseguir acesso de superusuário a partir de uma conta local do que fazer uma invasão remotamente. Com efeito, dada a gigantesca base de software (e os bugs resultantes e as falhas de design) disponível na maioria dos sistemas, pode ser fácil conseguir acesso de superusuário a um sistema depois que você obtiver um prompt de shell. Não suponha que seus amigos saibam como proteger suas senhas (ou saibam escolher boas senhas, antes de tudo).
- Evite instalar pacotes binários duvidosos – eles podem conter cavalos de Troia.

Essa é a parte prática no que diz respeito a se proteger. Mas por que é importante fazer isso? Há três tipos básicos de ataques a redes:

- Comprometimento total – quer dizer conseguir acesso de superusuário (controle completo) em um computador. Um invasor pode conseguir isso tentando executar um ataque a um serviço, por exemplo, explorar um buffer overflow (transbordamento de buffer), ou assumindo o controle de uma conta de usuário com uma proteção ineficiente e, em seguida, tentando explorar um programa setuid escrito de maneira inadequada.
- Ataque DoS (Denial-of-service, ou Negação de serviço) – esse ataque impede que um computador execute seus serviços de rede ou força um computador a ter um mau funcionamento de outra maneira, sem o uso de qualquer acesso especial. Esses ataques são mais difíceis de evitar, porém é mais fácil responder a eles.
- Malware – os usuários de Linux em sua maioria são imunes a malwares como worms e vírus de email simplesmente porque seus clientes de email não são estúpidos o suficiente para realmente executar os programas que vêm em anexos de mensagens. Porém os malwares Linux existem. Evite fazer download e instalar binários de software de locais dos quais você nunca ouviu falar.

10.7.1 Vulnerabilidades comuns

Há dois tipos importantes de vulnerabilidade com os quais devemos nos preocupar: ataques diretos e sniffing de senhas em formato texto simples. Os ataques diretos tentam assumir o controle de um computador sem muita sutileza. O ataque mais comum é a exploração de um buffer overflow, em que um programador descuidado não verifica os limites de um array de buffer. O invasor cria um stack frame dentro de uma porção grande de dados, descarrega esses dados em um servidor remoto e então espera que o servidor sobrescreva os dados de seu programa e que, em algum momento, execute a nova stack frame. Embora, de certo modo, seja um ataque complicado, ele é fácil de ser reproduzido.

Um segundo ataque com que devemos nos preocupar é aquele que captura senhas enviadas pela rede como texto simples. Assim que um invasor obtiver sua senha, o jogo chega ao fim. A partir daí, o invasor inevitavelmente tentará conseguir acesso de superusuário localmente (o que é muito mais fácil do que fazer um ataque remoto), tentará usar o computador como intermediário para atacar outros hosts, ou usará ambas as opções.

👍 **Observação:** se você tiver um serviço que não ofereça nenhum suporte nativo para criptografia, experimente usar o Stunnel (<http://www.stunnel.org/>) – um pacote wrapper para criptografia muito semelhante aos wrappers TCP. Assim como o tcpd, o Stunnel é especialmente adequado para wrapping de serviços inetd.

Alguns serviços são alvos crônicos de ataques por causa da implementação e do design pobres. Desative sempre os serviços a seguir (eles são raramente ativados por padrão na maioria dos sistemas):

- `ftpd` – por algum motivo, todos os servidores FTP parecem estar infestados de vulnerabilidades. Além disso, a maioria dos servidores FTP usa senhas em formato texto simples. Se houver necessidade de transferir arquivos de um computador para outro, considere uma solução baseada em SSH ou um servidor `rsync`.
- `telnetd`, `rlogind`, `rexecd` – todos eles passam dados de sessão remota (incluindo senhas) em formato texto simples. Evite-os, a menos que você tenha uma versão que permita usar Kerberos.
- `fingerd` – os invasores podem obter listas de usuários e outras informações usando o serviço `finger`.

10.7.2 Recursos relacionados à segurança

A seguir, apresentamos alguns bons sites relacionados à segurança:

- <http://www.sans.org/> – oferece treinamento, serviços, uma newsletter semanal gratuita que lista as principais vulnerabilidades do momento, exemplos de políticas de segurança e outras informações.
- <http://www.cert.org/> – um local para procurar os problemas mais graves.
- <http://www.insecure.org/> – é o local a ser acessado para buscar informações sobre o Nmap e referências para todos os tipos de ferramentas para testar exploits de rede. É um site muito mais aberto e específico sobre exploits do que vários outros sites.

Se você estiver interessado em segurança de redes, aprenda tudo sobre TLS (Transport Layer Security, ou Segurança da camada de transporte) e seu antecessor, o SSL (Secure Socket Layer, ou Camada segura de sockets). Esses níveis de rede do espaço de usuário normalmente são adicionados aos clientes e servidores de rede para suportar transações de rede por meio do uso de criptografia de chave pública e de certificados. Um bom guia é *Implementing SSL/TLS Using Cryptography and PKI* (Wiley, 2011) de Davies.

10.8 Próximos passos

Se você estiver interessado em colocar a mão na massa com alguns servidores de rede complexos, dois muito comuns são o servidor web Apache e o servidor de emails Postfix. Em particular, o Apache é fácil de instalar e a maioria das distribuições disponibiliza um pacote. Se o seu computador estiver atrás de um firewall ou de um roteador com NAT habilitado, você poderá fazer quantas experiências com a configuração que você quiser sem se preocupar com segurança.

Ao longo dos últimos capítulos, passamos gradualmente do espaço do kernel para o espaço do usuário. Somente alguns utilitários discutidos neste capítulo, como o `tcpdump`, interagem com o kernel. O restante deste capítulo descreve como os sockets preenchem as lacunas entre a camada de transporte do kernel e a camada de aplicação do espaço de usuário. É um material mais avançado, de interesse particular aos programadores, portanto sinta-se à vontade para pular para o próximo capítulo se quiser.

10.9 Sockets: como os processos se comunicam com a rede

Agora iremos mudar um pouco o foco e dar uma olhada em como os processos realizam a tarefa de ler e escrever dados na rede. Para os processos, é fácil ler e escrever dados

nas conexões de rede que já estejam estabelecidas: tudo o que é necessário são algumas chamadas de sistema, sobre as quais você pode ler nas páginas de manual `recv(2)` e `send(2)`. Do ponto de vista de um processo, talvez o aspecto mais importante a saber seja como se referir à rede ao usar essas chamadas de sistema. Nos sistemas Unix, um processo utiliza um *socket* para identificar quando e como ele conversa com a rede. Os sockets correspondem à interface que os processos usam para acessar a rede por meio do kernel; eles representam a fronteira entre o espaço de usuário e o espaço do kernel. Geralmente, os sockets também são usados para IPC (InterProcess Communication, ou Comunicação entre processos).

Há diferentes tipos de sockets, pois os processos devem acessar a rede de maneiras distintas. Por exemplo, as conexões TCP são representadas por stream sockets (sockets de stream ou `SOCK_STREAM`, do ponto de vista de um programador), e as conexões UDP são representadas por datagram sockets (sockets de datagrama, ou `SOCK_DGRAM`).

Configurar um socket de rede pode ser um tanto quanto complicado porque você deverá levar em consideração o tipo do socket, os endereços IP, as portas e o protocolo de transporte em determinados instantes. Entretanto, depois que todos os detalhes iniciais forem resolvidos, os servidores utilizarão determinados métodos-padrão para lidar com o tráfego de entrada da rede.

O diagrama de fluxo da figura 10.1 mostra como muitos servidores tratam conexões para stream sockets de entrada. Observe que esse tipo de servidor envolve dois tipos de socket: um socket que ouve e um socket para leitura e escrita. O processo principal usa o socket que ouve para procurar conexões na rede.

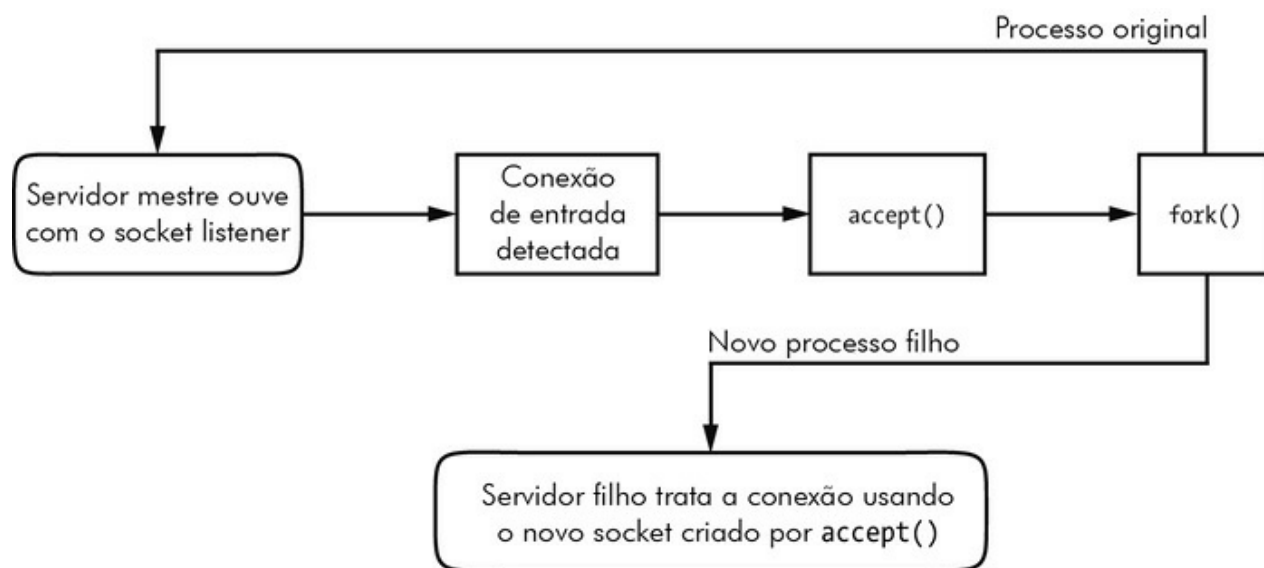


Figura 10.1 – Um método para aceitar e processar conexões de entrada.

Quando uma nova conexão surge, o processo principal utiliza a chamada de sistema

`accept()` para aceitar a conexão, o que cria um socket de leitura/escrita dedicado a essa conexão. Em seguida, o processo principal utiliza `fork()` para criar um novo processo-filho que cuidará da conexão. Por fim, o socket original permanece como listener e continua a esperar mais conexões em nome do processo principal.

Depois que um processo tiver configurado um socket de um determinado tipo, ele poderá interagir com ele de maneira que seja adequada ao tipo do socket. É isso que torna os sockets flexíveis: se houver necessidade de mudar a camada de transporte subjacente, não será preciso reescrever todas as partes que enviam e que recebem dados; em sua maior parte, você precisará modificar somente o código de inicialização.

Se você é programador e quer aprender a usar a interface de socket, o livro *Unix Network Programming, Volume 1*, 3ª edição, de W. Richard Stephens, Bill Fenner e Andrew M. Rudoff (Addison-Wesley Professional, 2003) é um guia clássico. O volume 2 também discute a comunicação entre processos.

10.10 Sockets de domínio Unix

As aplicações que usam recursos de rede não precisam envolver dois hosts diferentes. Muitas aplicações são criadas como cliente-servidor ou sistemas peer-to-peer (ponto a ponto), em que os processos que executam no mesmo computador usam IPC (InterProcess Communication, ou Comunicação entre processos) para negociar o trabalho a ser feito e quem deverá fazê-lo. Por exemplo, lembre-se de que daemons como o `systemd` e o `NetworkManager` usam D-Bus para monitorar e reagir a eventos do sistema.

Os processos podem usar redes IP normais sobre o localhost (127.0.0.1) para se comunicarem, porém, em vez disso, geralmente eles usam um tipo especial de socket, que discutimos brevemente no capítulo 3, chamado *sockets de domínio Unix*. Quando um processo se conecta a um socket de domínio Unix, ele se comporta quase exatamente como um socket de rede: ele pode ficar ouvindo à espera de conexões e aceitá-la no socket, e você pode até mesmo escolher entre diferentes tipos de socket para fazê-lo se comportar como TCP ou UDP.

👉 **Observação:** é importante se lembrar de que um socket de domínio Unix não é um socket de rede, e não há nenhuma rede por trás de um. Não é preciso nem mesmo que a rede esteja configurada para usar um socket desse tipo. E os sockets de domínio Unix não precisam estar associados a arquivos de socket. Um processo pode criar um socket de domínio Unix sem nome e compartilhar o endereço com outro processo.

10.10.1 Vantagens para os desenvolvedores

Os desenvolvedores gostam de sockets de domínio Unix para IPC por dois motivos. Em

primeiro lugar, eles permitem que os desenvolvedores tenham a opção de usar arquivos especiais de socket no sistema de arquivos para controlar o acesso de modo que qualquer processo que não tenha acesso a um arquivo de socket não poderá usá-lo. E como não há nenhuma interação com a rede, é uma solução mais simples e menos suscetível a invasões convencionais de rede. Por exemplo, você encontrará normalmente o arquivo de socket para D-Bus em */var/run/dbus*:

```
$ ls -l /var/run/dbus/system_bus_socket
srwxrwxrwx 1 root root 0 Nov  9 08:52 /var/run/dbus/system_bus_socket
```

Em segundo lugar, como o kernel do Linux não precisa passar pelas várias camadas de seu subsistema de rede ao trabalhar com sockets de domínio Unix, o desempenho tende a ser muito melhor.

Escrever código para sockets de domínio Unix não é muito diferente de suportar sockets de rede normais. Como as vantagens podem ser significativas, alguns servidores de rede oferecem comunicação tanto por meio de rede quanto por meio de sockets de domínio Unix. Por exemplo, o servidor de banco de dados MySQL `mysqld` pode aceitar conexões de cliente de hosts remotos, porém, normalmente, ele também disponibiliza um socket de domínio Unix em */var/run/mysqld/mysqld.sock*.

10.10.2 Listando os sockets de domínio Unix

Você pode ver uma lista de sockets de domínio Unix atualmente em uso em seu sistema usando o comando `lsof -U`:

```
# lsof -U
COMMAND  PID    USER  FD  TYPE  DEVICE  SIZE/OFF  NODE NAME
mysqld   19701  mysql  12u  unix 0xe4defcc0  0t0 35201227 /var/run/mysqld/mysqld.sock
chromium- 26534  juser   5u  unix 0xeeac9b00  0t0 42445141 socket
tlsmgr   30480  postfix 5u  unix 0xc3384240  0t0 17009106 socket
tlsmgr   30480  postfix 6u  unix 0xe20161c0  0t0 10965 private/tlsmgr
--trecho omitido--
```

A listagem será bem longa, pois muitas aplicações modernas fazem uso intensivo de sockets sem nome. É possível identificar os sockets sem nome porque você verá `socket` na coluna `NAME` da saída.

CAPÍTULO 11

Introdução aos shell scripts

Se você puder dar comandos no shell, poderá criar shell scripts (também conhecidos como Bourne shell scripts). Um *shell script* é uma série de comandos escritos em um arquivo; o shell lê os comandos do arquivo como faria se você os tivesse digitado em um terminal.

11.1 Básico sobre shell scripts

Os Bourne shell scripts geralmente começam com a linha a seguir, que indica que o programa `/bin/sh` deve executar os comandos que estão no arquivo de script. (Certifique-se de que não haverá nenhum espaço em branco no início do arquivo de script.)

```
#!/bin/sh
```

A parte referente a `#!` é conhecida como *shebang*; você a verá em outros scripts deste livro. Qualquer comando que você quiser que o shell execute poderá ser listado após a linha `#!/bin/sh`. Por exemplo:

```
#!/bin/sh
#
# Exibe uma mensagem e, em seguida, executa ls
echo About to run the ls command.
ls
```

 **Observação:** um caractere `#` no início de uma linha indica que a linha é um comentário, ou seja, o shell irá ignorar tudo o que estiver na linha após um `#`. Use comentários para explicar partes de seus scripts que sejam difíceis de entender.

Depois de ter criado um shell script e de ter configurado suas permissões, você poderá executá-lo colocando o arquivo em um dos diretórios que estejam em seu path de comandos e, em seguida, executando o nome do script na linha de comando. Você também pode executar `./script` se o script estiver em seu diretório de trabalho corrente, ou poderá usar o nome do path completo.

Como ocorre com qualquer programa em sistemas Unix, é preciso ligar o bit de executável para um arquivo de shell script, porém também será preciso ligar o bit de leitura para que o shell leia o arquivo. A maneira mais fácil de fazer isso é a seguinte:


```
$ chmod +rx script
```

O comando `chmod` permite que outros usuários leiam e executem *script*. Se não quiser fazer isso, utilize o modo absoluto `700` como alternativa (e consulte a seção 2.17 para relembrar as permissões).

Agora que já vimos o básico, vamos dar uma olhada em algumas das limitações dos shell scripts.

11.1.1 Limitações dos shell scripts

O Bourne shell manipula comandos e arquivos com relativa facilidade. Na seção 2.14, vimos a maneira pela qual o shell pode redirecionar a saída – um dos elementos importantes da programação de shell scripts. Entretanto o shell script é somente uma ferramenta de programação Unix e, embora os scripts tenham uma eficácia considerável, eles também têm limitações.

Um dos principais pontos fortes dos shell scripts é que eles podem simplificar e automatizar tarefas que, de outro modo, seriam realizadas no prompt de shell, por exemplo, manipular lotes de arquivos. Porém, se você estiver tentando separar strings, realizar processamentos aritméticos repetitivos ou acessar bancos de dados complexos, ou se quiser usar funções e estruturas de controle complexas, será melhor usar uma linguagem de scripting como Python, Perl ou `awk`, ou quem sabe até mesmo uma linguagem compilada como C. (Essa informação é importante, portanto iremos repeti-la ao longo do capítulo.)

Por fim, tome cuidado com o tamanho de seus shell scripts. Mantenha-os compactos. Os Bourne shell scripts não foram projetados para serem grandes (embora, sem dúvida, você vá se deparar com algumas monstruosidades).

11.2 Uso de aspas e literais

Um dos elementos mais confusos ao trabalhar com o shell e com os scripts é saber quando usar aspas (*quotes*) e outros sinais de pontuação, e por que às vezes é necessário fazer isso. Vamos supor que você queira exibir a string `$100` e faça o seguinte:

```
$ echo $100
00
```

Por que esse comando exibiu `00`? Porque o shell viu `$1`, que é uma variável de shell (discutiremos esse assunto em breve). Portanto você poderá pensar que, se colocar aspas duplas em torno do valor, o shell não irá se importar com o `$1`. Porém isso

continuará não funcionando:

```
$ echo "$100"  
00
```

Então você perguntará a um amigo, que dirá que é preciso usar aspas simples em vez das aspas duplas:

```
$ echo '$100'  
$100
```

Por que esse truque em particular funciona?

11.2.1 Literais

Ao usar aspas, geralmente você estará tentando criar um *literal* – uma string que você quer que o shell passe à linha de comando sem alterações. Além do \$ no exemplo que acabamos de ver, outras circunstâncias semelhantes incluem o caso em que você quer passar um caractere * a um comando como `grep`, em vez de fazer com que o shell o expanda, e quando for necessário usar um ponto e vírgula (;) em um comando.

Ao criar scripts e trabalhar na linha de comando, basta se lembrar do que acontece sempre que o shell executa um comando:

1. Antes de executar o comando, o shell procura variáveis, globs e outras substituições e as realiza caso ocorram.
2. O shell passa os resultados das substituições para o comando.

Os problemas que envolvem literais podem ser sutis. Vamos supor que você esteja procurando todas as entradas em `/etc/passwd` que correspondam à expressão regular `r.*t` (ou seja, uma linha que contenha um `r`, seguido de uma letra `t` mais adiante na linha, que possibilitará pesquisar nomes de usuário como `root`, `ruth` e `robot`). O comando a seguir pode ser executado:

```
$ grep r.*t /etc/passwd
```

Na maioria das vezes, ele funcionará, porém, às vezes, o comando falhará misteriosamente. Por quê? A resposta provavelmente está em seu diretório corrente. Se esse diretório contiver arquivos com nomes como `r.input` e `r.output`, o shell expandirá `r.*t` para `r.input r.output` e criará o comando a seguir:

```
$ grep r.input r.output /etc/passwd
```

O segredo para evitar problemas como esse é inicialmente reconhecer os caracteres que possam trazer problemas e, em seguida, aplicar o tipo correto de aspas para protegê-los.

11.2.2 Aspas simples

A maneira mais fácil de criar um literal e fazer com que o shell deixe a string intocada é colocar a string toda entre aspas simples, como no exemplo a seguir com `grep` e o caractere `*`:

```
$ grep 'r.*t' /etc/passwd
```

No que concerne ao shell, todos os caracteres entre as duas aspas simples, incluindo os espaços, compõem um único parâmetro. Sendo assim, o comando a seguir *não* funcionará, pois ele pede ao comando `grep` que procure a string `r.*t /etc/passwd` na saída-padrão (porque há somente um parâmetro para o `grep`):

```
$ grep 'r.*t /etc/passwd'
```

Quando houver necessidade de usar um literal, você sempre deverá lançar mão das aspas simples antes, pois terá a garantia de que o shell não tentará fazer *nenhuma* substituição. Como resultado, a sintaxe, em geral, será clara. Às vezes, porém, será preciso ter um pouco mais de flexibilidade; nesse caso, você poderá lançar mão das aspas duplas.

11.2.3 Aspas duplas

As aspas duplas (`"`) funcionam como as aspas simples, exceto pelo fato de o shell expandir qualquer variável que apareça entre elas. Você pode perceber a diferença ao executar o comando a seguir e, em seguida, substituindo as aspas duplas por aspas simples e executando-o novamente.

```
$ echo "There is no * in my path: $PATH"
```

Ao executar o comando, observe que o shell faz substituições para `$PATH`, mas não para `*`.

 **Observação:** se você estiver usando aspas duplas ao exibir uma grande quantidade de texto, considere usar um *here document*, conforme descrito na seção 11.9.

11.2.4 Passando aspas simples literais

Uma parte intrincada do uso de literais com o Bourne shell surge quando queremos passar aspas simples literais a um comando. Uma maneira de fazer isso é colocando uma barra invertida antes do caractere de aspas simples:

```
$ echo I don\'t like contractions inside shell scripts.
```

A barra invertida e o sinal de aspas *devem* aparecer fora de qualquer par de aspas simples, e uma string como `'don\'t` resulta em erro de sintaxe. Por mais estranho que possa parecer, podemos colocar as aspas simples dentro de aspas duplas, como mostrado no

exemplo a seguir (a saída será idêntica àquela do comando anterior):

```
$ echo "I don't like contractions inside shell scripts."
```

Se você estiver em uma situação difícil e precisar de uma regra geral para colocar uma string toda entre aspas, sem substituições, siga o procedimento a seguir:

1. Mude todas as instâncias de ' (aspas simples) para \" (aspas simples, barra invertida, aspas simples, aspas simples).
2. Coloque a string toda entre aspas simples.

Desse modo, você poderá usar aspas em strings inusitadas como this isn't a forward slash: \ da seguinte maneira:

```
$ echo 'this isn\'t a forward slash: \'
```

 **Observação:** vale a pena repetir que ao colocar uma string entre aspas, o shell tratará tudo o que estiver entre elas como um único parâmetro. Sendo assim, a b c é considerado como três parâmetros, porém a "b c" representa somente dois.

11.3 Variáveis especiais

A maioria dos shell scripts entende parâmetros de linha de comando e interage com os comandos que executam. Para fazer com que seus scripts deixem de ser uma simples lista de comandos e se transformem em programas flexíveis de shell script, é preciso saber usar as variáveis especiais do Bourne shell. Essas variáveis especiais são como qualquer outra variável de shell, conforme descritas na seção 2.8, exceto pelo fato de você não poder alterar o valor de determinadas variáveis.

 **Observação:** após ler as próximas seções, você entenderá por que os shell scripts acumulam vários caracteres especiais à medida que são escritos. Se você estiver tentando entender um shell script e se deparar com uma linha que pareça totalmente incompreensível, separe-a por partes.

11.3.1 Argumentos individuais: \$1, \$2, ...

\$1, \$2 e todas as variáveis nomeadas como inteiros positivos diferentes de zero contêm os valores de parâmetros do script, ou argumentos. Por exemplo, suponha que o nome do script a seguir seja *pshow*:

```
#!/bin/sh
echo First argument: $1
echo Third argument: $3
```

Experimente executar o script como se segue para ver como os argumentos são exibidos:

```
$ ./pshow one two three
```

First argument: one
Third argument: three

O comando `shift` já incluído no shell pode ser usado com variáveis de argumento para remover o primeiro argumento (`$1`) e fazer avanços no restante deles. Especificamente, `$2` se torna `$1`, `$3` se torna `$2` e assim por diante. Por exemplo, suponha que o nome do script a seguir seja *shiftext*:

```
#!/bin/sh
echo Argument: $1
shift
echo Argument: $1
shift
echo Argument: $1
```

Execute-os do modo como se segue para ver como ele funciona:

```
$ ./shiftext one two three
Argument: one
Argument: two
Argument: three
```

Como você pode ver, *shiftext* exibe todos os três argumentos mostrando o primeiro, deslocando os argumentos restantes e repetindo o processo.

11.3.2 Número de argumentos: `$#`

A variável `$#` armazena a quantidade de argumentos passada para um script e é particularmente importante quando executamos `shift` em um laço para percorrer os argumentos. Quando `$#` for igual a 0, não haverá mais argumentos, portanto `$1` estará vazio. (Consulte a seção 11.6 para ver uma descrição dos laços.)

11.3.3 Todos os argumentos: `$@`

A variável `$@` representa todos os argumentos de um script, e é muito útil para passá-los a um comando do script. Por exemplo, os comandos Ghostscript (`gs`) geralmente são longos e complicados. Suponha que você queira ter um atalho para rasterizar um arquivo PostScript em 150 dpi, usando a stream de saída-padrão, ao mesmo tempo que deixa a porta aberta para passar outras opções ao `gs`. Você pode criar um script como o que se segue para permitir que haja opções adicionais na linha de comando:

```
#!/bin/sh
gs -q -dBATCH -dNOPAUSE -dSAFER -sOutputFile=- -sDEVICE=psmraw $@
```

 **Observação:** se uma linha de seu shell script se tornar longa demais para seu editor de texto, ela poderá ser dividida com uma barra invertida (`\`). Por exemplo, o script anterior pode ser alterado da seguinte maneira:

```
#!/bin/sh
gs -q -dPATCH -dNOPAUSE -dSAFER \
-sOutputFile=- -sDEVICE=pnmraw $@
```

11.3.4 Nome do script: \$0

A variável \$0 contém o nome do script e é útil para gerar mensagens de diagnóstico. Por exemplo, suponha que seu script precise informar um argumento inválido que esteja armazenado na variável \$BADPARAM. A mensagem de diagnóstico poderá ser exibida com a linha a seguir, de modo que o nome do script apareça na mensagem de erro:

```
echo $0: bad option $BADPARAM
```

Todas as mensagens de diagnóstico devem ir para o erro-padrão. Lembre-se de que, conforme descrito na seção 2.14.1, 2>&1 redireciona o erro-padrão para a saída-padrão. Para escrever na saída-padrão, o processo pode ser invertido com 1>&2. Para fazer isso no exemplo anterior, utilize:

```
echo $0: bad option $BADPARAM 1>&2
```

11.3.5 ID do processo: \$\$

A variável \$\$ armazena o ID do processo relacionado ao shell.

11.3.6 Código de saída: \$?

A variável \$? armazena o código de saída do último comando executado pelo shell. Os códigos de saída, que são cruciais para dominar os shell scripts, serão discutidos a seguir.

11.4 Códigos de saída

Quando um programa Unix termina, ele deixa um *código de saída* para o processo-pai que o iniciou. O código de saída é um número e, às vezes, é chamado de *código de erro* ou de *valor de saída*. Quando o código de saída for igual a zero (0), é sinal de que o programa executou sem problemas. Contudo, se o programa tiver um erro, normalmente ele terminará com um número diferente de 0 (mas nem sempre, como você verá a seguir).

O shell armazena o código de saída do último comando na variável especial \$? para que você possa verificá-la em seu prompt de shell:

```
$ ls /> /dev/null
$ echo $?
0
```

```
$ ls /asdfasdf > /dev/null
ls: /asdfasdf: No such file or directory
$ echo $?
1
```

Você pode perceber que o comando bem-sucedido retornou 0 e que o comando sem sucesso retornou 1 (supondo, é claro, que você não tenha um diretório chamado */asdfasdf* em seu sistema).

Se você pretende usar o código de saída de um comando, você *deverá* usar ou armazenar o código imediatamente após executar o comando. Por exemplo, se `echo $?` for executado duas vezes seguidas, a saída do segundo comando será sempre igual a 0, pois o primeiro comando `echo` termina com sucesso.

Ao escrever um código de shell que não encerre um script normalmente, use algo como `exit 1` para passar um código de saída igual a 1 de volta a qualquer que seja o processo-pai que tenha executado o script. (Você pode usar números diferentes para condições diferentes.)

Um aspecto a ser observado é que alguns programas como `diff` e `grep` usam códigos de saída diferentes de zero para indicar condições normais. Por exemplo, `grep` retorna 0 se encontrar algo que corresponda a um padrão, e 1, caso contrário. Para esses programas, um código de saída igual a 1 não é um erro; `grep` e `diff` usam o código de saída 2 para problemas de verdade. Se você acha que um programa está usando um código de saída diferente de zero para indicar sucesso, leia sua página de manual. Os códigos de saída normalmente são explicados na seção `EXIT VALUE` (Valor de saída) ou `DIAGNOSTICS` (Diagnósticos).

11.5 Condicionais

O Bourne shell tem construções especiais para condicionais, como instruções `if/then/else` e `case`. Por exemplo, o script simples a seguir com uma condicional `if` verifica se o primeiro argumento do script é `hi`:

```
#!/bin/sh
if [ $1 = hi ]; then
    echo "The first argument was "hi"
else
    echo -n "The first argument was not "hi" -- "
    echo It was ""$1""
fi
```

As palavras `if`, `then`, `else` e `fi` no script anterior são palavras-chave do shell; o restante são

comandos. A distinção é extremamente importante, pois um dos comandos é `[ $1 = "hi" ]` e o caractere `[` na verdade é um programa em um sistema Unix, e *não* uma sintaxe especial do shell. (Isso não é bem verdade, como você logo aprenderá, porém trate-o como um comando separado em sua mente, por enquanto.) Todos os sistemas Unix têm um comando chamado `[` para realizar testes para condicionais em shell script. Esse programa também é conhecido como `test`, e uma análise cuidadosa de `[` e `test` devem revelar que eles compartilham um inode, ou que um deles é um link simbólico para o outro.

Entender os códigos de saída da seção 11.4 é fundamental, pois o processo como um todo funciona da seguinte maneira:

1. O shell executa o comando depois da palavra-chave `if` e obtém o código de saída desse comando.
2. Se o código de saída for igual a 0, o shell executará os comandos que vierem depois da palavra-chave `then`, parando quando alcançar a palavra-chave `else` ou `fi`.
3. Se o código de saída não for igual a 0 e houver uma cláusula `else`, o shell executará os comandos após a palavra-chave `else`.
4. A condicional termina em `fi`.

11.5.1 Lidando com listas de parâmetros vazias

Há um pequeno problema com a condicional do exemplo anterior em consequência de um erro bastante comum: `$1` poderia estar vazia, pois o usuário poderia não ter fornecido um parâmetro. Sem um parâmetro, o teste seria `[ = hi ]`, e o comando `[` seria abortado com um erro. Isso pode ser corrigido ao colocar o parâmetro entre aspas usando uma entre duas opções (ambas são comuns):

```
if [ "$1" = hi ]; then
if [ x"$1" = x"hi" ]; then
```

11.5.2 Usando outros comandos para testes

O que vier depois de `if` será sempre um comando. Sendo assim, se você quiser colocar a palavra-chave `then` na mesma linha, é preciso usar um ponto e vírgula (;) após o comando de teste. Se o ponto e vírgula não for usado, o shell passará `then` como parâmetro para o comando de teste. (Se você não gosta do ponto e vírgula, a palavra-chave `then` poderá ser colocada em uma linha separada.)

Há várias possibilidades para usar outros comandos no lugar de `[`. A seguir, apresentamos um exemplo que usa `grep`:


```
#!/bin/sh
if grep -q daemon /etc/passwd; then
    echo The daemon user is in the passwd file.
else
    echo There is a big problem. daemon is not in the passwd file.
fi
```

11.5.3 elif

Há também uma palavra-chave `elif` que permite encadear condicionais `if`, como mostrado a seguir. Porém não fique muito empolgado com o `elif` porque a construção `case` que você verá na seção 11.5.6 geralmente será mais apropriada.

```
#!/bin/sh
if [ "$1" = "hi" ]; then
    echo 'The first argument was "hi"'
elif [ "$2" = "bye" ]; then
    echo 'The second argument was "bye"'
else
    echo -n 'The first argument was not "hi" and the second was not "bye"-- '
    echo They were ""$1"" and ""$2""
fi
```

11.5.4 Construções lógicas `&&` e `||`

Há duas construções condicionais rápidas de uma só linha que você poderá ver ocasionalmente: `&&` (“and”) e `||` (“or”). A construção `&&` funciona da seguinte maneira:

comando1 && comando2

Nesse caso, o shell executa *comando1* e, se o código de saída for igual a 0, o shell executará também *comando2*. A construção `||` é semelhante; se o comando antes de `||` retornar um código de saída diferente de zero, o shell executará o segundo comando.

As construções `&&` e `||` geralmente são usadas em testes `if` e, em ambos os casos, o código de saída do último comando executado determina o modo como o shell processará a condicional. No caso da construção `&&`, se o primeiro comando falhar, o shell usará o seu código de saída para a instrução `if`; porém, se o primeiro comando for bem-sucedido, o shell usará o código de saída do segundo comando para a condicional. No caso da construção `||`, o shell usará o código de saída do primeiro comando se esse for bem-sucedido, ou o código de saída do segundo se o primeiro não tiver sucesso.

Por exemplo:

```
#!/bin/sh
if [ "$1" = hi ] || [ "$1" = bye ]; then
```

```
echo 'The first argument was "$1"'
fi
```

Se suas condicionais incluírem o comando de teste ([]), como mostrado nesse caso, você poderá usar -a e -o no lugar de && e ||, conforme será descrito na próxima seção.

11.5.5 Condições de teste

Você viu como [] funciona: o código de saída será 0 se o teste for verdadeiro e diferente de zero se o teste falhar. Você também sabe testar igualdade de strings com [*str1* = *str2*]. No entanto, lembre-se de que os shell scripts são bastante adequados para operações em arquivos inteiros porque os testes [] mais úteis envolvem propriedades de arquivo. Por exemplo, a linha a seguir verifica se *arquivo* é um arquivo normal (e não um diretório ou um arquivo especial):

```
[ -f arquivo ]
```

Em um script, você poderá ver o teste -f em um laço semelhante ao que está sendo mostrado a seguir, que testa todos os itens do diretório de trabalho corrente (você aprenderá mais sobre laços em geral em breve):

```
for filename in *; do
    if [ -f $filename ]; then
        ls -l $filename
        file $filename
    else
        echo $filename is not a regular file.
    fi
done
```

Um teste pode ser invertido se o operador ! for colocado antes dos argumentos do teste. Por exemplo, [! -f *arquivo*] retornará verdadeiro se *arquivo* não for um arquivo normal. Além do mais, as flags -a e -o correspondem aos operadores lógicos “and” (e) e “or” (ou) (por exemplo, [-f *arquivo1* -a *arquivo2*]).

 **Observação:** pelo fato de o comando test ser tão amplamente usado em scripts, muitas versões do Bourne shell (incluindo bash) já incluem o comando test. Isso pode agilizar os scripts, pois o shell não precisará executar um comando separado para cada teste.

Há dezenas de operações de teste, e todas elas se enquadram em três categorias gerais: testes de arquivo, testes de string e testes aritméticos. O manual info contém uma documentação online completa, porém a página de manual test(1) contém uma referência rápida. As seções a seguir descrevem os principais testes. (Omiti alguns dos testes menos comuns.)

Testes de arquivo

A maioria dos testes de arquivo, como `-f`, é chamada de operações *unárias* porque exigem somente um argumento: o arquivo a ser testado. Por exemplo, a seguir apresentamos dois testes importantes de arquivo:

- `-e` – retorna verdadeiro se um arquivo existir.
- `-s` – retorna verdadeiro se um arquivo não estiver vazio.

Várias operações inspecionam o tipo de um arquivo, o que significa que elas podem determinar se um item é um arquivo normal, um diretório ou algum tipo de dispositivo especial, conforme listado na tabela 11.1. Há também diversas operações unárias que verificam as permissões de um arquivo, conforme listado na tabela 11.2. (Consulte a seção 2.17 para ter uma visão geral das permissões.)

Tabela 11.1 – Operadores relacionados a tipos de arquivo

Operador	Teste para
<code>-f</code>	Arquivo normal
<code>-d</code>	Diretório
<code>-h</code>	Link simbólico
<code>-b</code>	Dispositivo de bloco
<code>-c</code>	Dispositivo de caractere
<code>-p</code>	Pipe nomeado
<code>-S</code>	Socket

 **Observação:** o comando `test` segue links simbólicos (exceto para o teste `-h`). Isso quer dizer que se *link* for um link simbólico para um arquivo normal, `[ -f link ]` retornará um código de saída igual a verdadeiro (0).

Tabela 11.2 – Operadores para permissão de arquivos

Operador	Permissão
<code>-r</code>	Leitura
<code>-w</code>	Escrita
<code>-x</code>	Executável
<code>-u</code>	Setuid
<code>-g</code>	Setgid
<code>-k</code>	“Sticky”

Por fim, três operadores *binários* (testes que exigem dois arquivos como argumentos) são usados em testes de arquivo, porém não são muito comuns. Considere o comando a seguir que inclui *-nt* (*newer than*, ou mais novo que):

```
[ arquivo1 -nt arquivo2 ]
```

Essa operação retorna verdadeiro se *arquivo1* tiver uma data de modificação mais recente que *arquivo2*. O operador *-ot* (*older than*, ou mais velho que) faz o contrário. E se você precisar detectar hard links idênticos, *-ef* compara dois arquivos e retorna verdadeiro se eles compartilharem números de inode e dispositivos.

Testes de string

Já vimos o operador binário de string = que retorna verdadeiro se os seus operandos forem iguais. O operador != retorna verdadeiro se os seus operandos não forem iguais. E há duas operações unárias de string:

- *-z* – retorna verdadeiro se o seu argumento for vazio ([*-z ""*] retorna 0).
- *-n* – retorna verdadeiro se o seu argumento não for vazio ([*-n ""*] retorna 1).

Testes aritméticos

É importante reconhecer que o sinal de igualdade (=) verifica igualdade de *strings*, e não igualdade *numérica*. Sendo assim, [*1 = 1*] retorna 0 (verdadeiro), porém [*01 = 1*] retorna falso. Ao trabalhar com números, utilize *-eq* no lugar do sinal de igualdade: [*01 -eq 1*] retorna verdadeiro. A tabela 11.3 apresenta a lista completa dos operadores de comparação numérica.

Tabela 11.3 – Operadores de comparação aritmética

Operador	Retorna verdadeiro quando o primeiro argumento for ... o segundo argumento
<i>-eq</i>	Igual a
<i>-ne</i>	Diferente de
<i>-lt</i>	Menor que
<i>-gt</i>	Maior que
<i>-le</i>	Menor ou igual a
<i>-ge</i>	Maior ou igual a

11.5.6 Fazendo a correspondência de strings com case

A palavra-chave *case* compõe outra construção condicional excepcionalmente útil para

correspondência de strings. A condicional `case` não execute nenhum comando de teste e, sendo assim, não avalia códigos de saída. Entretanto ela pode fazer correspondência com padrões. O exemplo a seguir deve contar a maior parte da história:

```
#!/bin/sh
case $1 in
  bye)
    echo Fine, bye.
    ;;
  hi|hello)
    echo Nice to see you.
    ;;
  what*)
    echo Whatever.
    ;;
  *)
    echo 'Huh?'
    ;;
esac
```

O shell executa isso da seguinte maneira:

1. O script faz a correspondência entre `$1` e cada valor de caso demarcado com o caractere `)`.
2. Se o valor de um caso corresponder a `$1`, o shell executará os comandos abaixo do caso até encontrar `;;` – nesse ponto, ele pulará para a palavra-chave `esac`.
3. A condicional termina com `esac`.

Para cada valor de caso, você poderá fazer a correspondência de uma única string (como `bye` no exemplo anterior) ou de várias strings com `|` (`hi|hello` retorna verdadeiro se `$1` for igual a `hi` ou a `hello`), ou os padrões `*` ou `?` (`what*`) podem ser usados. Para criar um caso default que capture todos os valores possíveis que não sejam os valores de caso especificados, utilize um único `*`, como mostrado no caso final do exemplo anterior.

 **Observação:** cada caso deve terminar com dois pontos e vírgulas (`;;`) ou você correrá o risco de ter um erro de sintaxe.

11.6 Laços

Há dois tipos de laço no Bourne shell: `for` e `while`.

11.6.1 Laços `for`

O laço `for` (que é um laço “for each”) é o mais comum. Aqui está um exemplo:

```
#!/bin/sh
for str in one two three four; do
    echo $str
done
```

Nessa listagem, `for`, `in`, `do` e `done` são palavras-chave do shell. O shell faz o seguinte:

1. Define a variável `str` com o primeiro dos quatro valores delimitados por espaço que se seguem à palavra-chave `in` (`one`).
2. Executa o comando `echo` entre `do` e `done`.
3. Retorna à linha `for`, definindo `str` com o próximo valor (`two`), executa os comandos entre `do` e `done` e repete o processo até ter passado por todos os valores que se seguem à palavra-chave `in`.

A saída desse script terá a seguinte aparência:

```
one
two
three
four
```

11.6.2 Laços `while`

O laço `while` do Bourne shell usa códigos de saída como a condicional `if`. Por exemplo, o script a seguir realiza dez iterações:

```
#!/bin/sh
FILE=/tmp/whiletest.$$;
echo firstline > $FILE
while tail -10 $FILE | grep -q firstline; do
    # adiciona linhas a $FILE até tail -10 $FILE não exibir mais "firstline"
    echo -n Number of lines in $FILE: '
    wc -l $FILE | awk '{print $1}'
    echo newline >> $FILE
done
rm -f $FILE
```

Nesse caso, o código de saída de `grep -q firstline` é o teste. Assim que o código de saída for diferente de zero (nesse caso, quando a string `firstline` não aparecer mais nas dez últimas linhas em `$FILE`), o laço terminará.

Você pode sair de um laço `while` usando a instrução `break`. O Bourne shell também tem um laço `until` que funciona exatamente como `while`, exceto pelo fato de sair do laço quando encontra um código de saída igual a zero, em vez de um código de saída diferente de zero. Apesar do que foi dito, você não deverá precisar usar os laços `while` e `until` com

muita frequência. Na verdade, se achar que precisa usar `while`, provavelmente você deverá usar uma linguagem como `awk` ou `Python` em seu lugar.

11.7 Substituição de comandos

O Bourne shell pode redirecionar a saída-padrão de um comando de volta para a própria linha de comando do shell. Ou seja, você pode usar a saída de um comando como argumento para outro comando, ou pode armazenar a saída do comando em uma variável de shell ao colocar o comando entre `$()`.

O exemplo a seguir armazena um comando na variável `FLAGS`. O negrito na segunda linha mostra a substituição do comando.

```
#!/bin/sh
FLAGS=$(grep ^flags /proc/cpuinfo | sed 's/.*/' | head -1)
echo Your processor supports:
for f in $FLAGS; do
    case $f in
        fpu) MSG="floating point unit"
            ;;
        3dnow) MSG="3DNow graphics extensions"
            ;;
        mtrr) MSG="memory type range register"
            ;;
        *) MSG="unknown"
            ;;
    esac
    echo $f: $MSG
done
```

Esse exemplo, de certo modo, é complicado, pois mostra que você pode usar tanto aspas simples quanto pipelines na substituição do comando. O resultado do comando `grep` é enviado ao comando `sed` (mais sobre o `sed` na seção 11.10.3), que remove tudo o que corresponder à expressão `.*`; e o resultado do `sed` será passado para `head`.

É fácil exagerar com a substituição de comandos. Por exemplo, não use `$(ls)` em um script porque usar o shell para expandir `*` é mais rápido. Além do mais, se você quiser chamar um comando em vários nomes de arquivo obtidos como resultado de um comando `find`, considere usar um pipeline para `xargs` em vez de usar a substituição de comandos, ou utilize a opção `-exec` (veja a seção 11.10.4).

 **Observação:** a sintaxe tradicional para substituição de comandos consiste em colocar o comando entre crases (```), e você verá isso em vários shell scripts. A sintaxe `$()` é um formato novo, porém é um padrão POSIX e, geralmente, é mais fácil de ler e de escrever.

11.8 Gerenciamento de arquivos temporários

Às vezes, é necessário criar um arquivo temporário para guardar resultados a serem usados por um comando mais tarde. Ao criar um arquivo desse tipo, certifique-se de que o nome do arquivo seja diferente o suficiente, de modo que nenhum outro programa escreva nele acidentalmente.

Eis o modo de usar o comando `mktemp` para criar nomes de arquivo temporários. O script a seguir mostra as interrupções de dispositivo que ocorreram nos dois últimos segundos:

```
#!/bin/sh
TMPFILE1=$(mktemp /tmp/im1.XXXXXXX)
TMPFILE2=$(mktemp /tmp/im2.XXXXXXX)

cat /proc/interrupts > $TMPFILE1
sleep 2
cat /proc/interrupts > $TMPFILE2
diff $TMPFILE1 $TMPFILE2
rm -f $TMPFILE1 $TMPFILE2
```

O argumento de `mktemp` é um template. O comando `mktemp` converte o `XXXXXX` para um conjunto único de caracteres e cria um arquivo vazio com esse nome. Observe que esse script usa nomes de variáveis para armazenar os nomes de arquivo de modo que você precisará alterar somente uma linha se quiser mudar um nome de arquivo.

 **Observação:** nem todas as variantes de Unix vêm com `mktemp`. Se você estiver tendo problemas de portabilidade, é melhor instalar o pacote GNU coreutils para o seu sistema operacional.

Outro problema com scripts que empregam arquivos temporários é que, se o script for abortado, os arquivos temporários poderão ser deixados para trás. No exemplo anterior, teclar `ctrl-C` antes do segundo comando `cat` deixará um arquivo temporário em `/tmp`. Evite isso se for possível. Utilize o comando `trap` para criar um handler de sinal para capturar o sinal gerado por `Ctrl-C` e remover os arquivos temporários, como ocorre no handler a seguir:

```
#!/bin/sh
TMPFILE1=$(mktemp /tmp/im1.XXXXXXX)
TMPFILE2=$(mktemp /tmp/im2.XXXXXXX)
trap "rm -f $TMPFILE1 $TMPFILE2; exit 1" INT
--trecho omitido--
```

Use `exit` no handler para finalizar explicitamente a execução do script, ou o shell continuará executando normalmente após executar o handler de sinal.

 **Observação:** não é preciso fornecer um argumento para `mktemp`; se você não o fizer, o template

começará com um prefixo `/tmp/tmp..`

11.9 Here documents

Suponha que você queira exibir uma seção grande de texto ou queira fornecer bastante texto para outro comando. Em vez de usar vários comandos `echo`, podemos usar o recurso de *here document* do shell, conforme mostrado no script a seguir:

```
#!/bin/sh
DATE=$(date)
cat <<EOF
Date: $DATE

The output above is from the Unix date command.
It's not a very interesting command.
EOF
```

Os itens em negrito controlam o here document. O `<<EOF` diz ao shell para redirecionar todas as linhas que se seguirem à entrada-padrão do comando que antecede `<<EOF`, que, nesse caso, é `cat`. O redirecionamento é interrompido assim que o marcador `EOF` ocorre em uma linha sozinho. O marcador pode ser qualquer string, na verdade, mas lembre-se de usar o mesmo marcador no início e no fim do here document. Além do mais, a convenção determina que o marcador tenha somente letras maiúsculas.

Observe a variável de shell `$DATE` no here document. O shell expande as variáveis de shell dentro dos here documents, o que será especialmente útil quando você estiver exibindo relatórios que contenham muitas variáveis.

11.10 Utilitários importantes para shell scripts

Vários programas são particularmente úteis em shell scripts. Determinados utilitários como `basename` são realmente práticos somente quando usados com outros programas e, desse modo, geralmente não encontram lugar fora dos shell scripts. No entanto outros como `awk` podem ser muito úteis na linha de comando também.

11.10.1 basename

Se você precisar remover a extensão de um nome de arquivo ou se livrar dos diretórios em um nome de path completo, utilize o comando `basename`. Experimente usar os exemplos a seguir na linha de comando para ver como o comando funciona:

```
$ basename example.html .html
$ basename /usr/local/bin/example
```

Em ambos os casos, `basename` retorna `example`. O primeiro comando remove o sufixo `.html` de `example.html`, e o segundo remove os diretórios do path completo.

O exemplo a seguir mostra como você pode usar `basename` em um script para converter arquivos de imagem GIF em formato PNG:

```
#!/bin/sh
for file in *.gif; do
    # sai se não houver arquivos
    if [ ! -f $file ]; then
        exit
    fi
    b=$(basename $file .gif)
    echo Converting $b.gif to $b.png...
    giftopnm $b.gif | pnmtopng > $b.png
done
```

11.10.2 awk

O comando `awk` não é um comando simples com um único propósito; na verdade, ele é uma linguagem de programação eficaz. Infelizmente, o uso do `awk` atualmente é uma espécie de arte perdida, tendo sido substituído por linguagens mais abrangentes como Python.

Há livros inteiros sobre o assunto `awk`, incluindo *The AWK Programming Language* de Alfred V. Aho, Brian W. Kernighan e Peter J. Weinberger (Addison-Wesley, 1988). Apesar do que foi dito, muitas e muitas pessoas usam o `awk` para realizar uma tarefa – escolher um único campo de um stream de entrada da seguinte maneira:

```
$ ls -l | awk '{print $5}'
```

Esse comando exibe o quinto campo da saída de `ls` (o tamanho do arquivo). O resultado é uma lista de tamanhos de arquivo.

11.10.3 sed

O programa `sed` (`sed` quer dizer stream editor, ou editor de stream) é um editor de texto automático que recebe um stream de entrada (um arquivo ou a entrada-padrão), altera-o de acordo com alguma expressão e exibe o resultado na saída-padrão. Em vários aspectos, o `sed` é como o `ed`, que é o editor de texto original do Unix. Ele tem dezenas de operações, ferramentas de correspondência e recursos de endereçamento. Assim como no caso do `awk`, livros inteiros foram escritos sobre o `sed`, incluindo uma referência rápida que inclui ambos – o livro *sed & awk Pocket Reference*, 2ª edição, de Arnold Robbins (O'Reilly, 2002).

Embora o `sed` seja um programa importante, e uma análise detalhada esteja além do escopo deste livro, é fácil ver como esse programa funciona. Em geral, o `sed` recebe um endereço e uma operação como argumento. O endereço corresponde a um conjunto de linhas, e o comando determina o que fazer com as linhas.

Uma tarefa bastante comum para o `sed` é substituir uma expressão regular por um texto (veja a seção 2.5.1), deste modo:

```
$ sed 's/exp/textol'
```

Portanto, se você quiser substituir o primeiro dois-pontos em `/etc/passwd` por um `%` e enviar o resultado para a saída-padrão, isso pode ser feito da seguinte maneira:

```
$ sed 's/:/%/' /etc/passwd
```

Para substituir *todos* os dois-pontos em `/etc/passwd`, adicione um modificador `g` ao final da operação, da seguinte maneira:

```
$ sed 's/:/%/g' /etc/passwd
```

A seguir, apresentamos um comando que opera por linha; ele lê `/etc/passwd`, apaga as linhas de três a seis e envia o resultado para a saída-padrão:

```
$ sed 3,6d /etc/passwd
```

Nesse exemplo, `3,6` é o endereço (um intervalo de linhas), e `d` é a operação (delete, ou apagar). Se o endereço for omitido, o `sed` atuará em todas as linhas de seu stream de entrada. As duas operações mais comuns de `sed` provavelmente são `s` (search and replace, ou pesquisar e substituir) e `d`.

Uma expressão regular também pode ser usada como endereço. O comando a seguir apaga qualquer linha que corresponda à expressão regular `exp`:

```
$ sed '/exp/d'
```

11.10.4 `xargs`

Quando for preciso executar um comando em uma quantidade enorme de arquivos, o comando ou o shell poderá responder que não é possível colocar todos os argumentos em seu buffer. Use `xargs` para contornar esse problema ao executar um comando em cada arquivo em seu stream de entrada-padrão.

Muitas pessoas usam `xargs` com o comando `find`. Por exemplo, o script a seguir pode ajudar a verificar se todos os arquivos na árvore do diretório corrente que terminem com `.gif` são realmente uma imagem GIF (Graphic Interchange Format):

```
$ find . -name '*.gif' -print | xargs file
```

No exemplo anterior, `xargs` executa o comando `file`. Entretanto essa chamada pode causar

erros ou deixar seu sistema sujeito a problemas de segurança, pois os nomes de arquivo podem incluir espaços ou quebras de linha. Ao escrever um script, use o formato a seguir, que muda o separador da saída de `find` e o delimitador de argumentos de `xargs`, transformando as quebras de linha em um caractere NULL:

```
$ find . -name '*.gif' -print0 | xargs -0 file
```

`xargs` inicia *vários* processos, portanto não espere um ótimo desempenho se você tiver uma lista longa de arquivos.

Talvez seja necessário adicionar dois hifens (`--`) no final de seu comando `xargs` se houver alguma chance de algum dos arquivos-alvos começar com um único hífen (`-`). Os dois hifens (`--`) podem ser usados para dizer a um programa que qualquer argumento que se seguir aos dois hifens é um nome de arquivo, e não uma opção. No entanto tenha em mente que nem todos os programas suportam o uso dos dois hifens.

Há uma alternativa ao `xargs` ao usar `find`: a opção `-exec`. Entretanto a sintaxe é um pouco complicada porque você deverá fornecer um `{}` para substituir o nome do arquivo e um literal `;` para indicar o final do comando. A seguir, apresentamos a maneira de realizar a tarefa anterior usando somente `find`:

```
$ find . -name '*.gif' -exec file {} \;
```

11.10.5 `expr`

Se for necessário usar operações aritméticas em seus shell scripts, o comando `expr` poderá ajudar (e até mesmo realizar algumas operações de string). Por exemplo, o comando `expr 1 + 2` exibe 3. (Execute `expr --help` para obter uma lista completa de operações.)

O comando `expr` é uma maneira desajeitada e lenta de realizar operações matemáticas. Se você estiver usando esse comando com muita frequência, provavelmente uma linguagem como Python deverá ser usada no lugar de um shell script.

11.10.6 `exec`

O comando `exec` é um recurso pronto do shell que substitui o processo corrente do shell pelo programa que você especificar após `exec`. Ele executa a chamada de sistema `exec()`, que você viu no capítulo 1. Esse recurso foi concebido para economizar recursos do sistema, porém lembre-se de que não há retorno; quando `exec` for executado em um shell script, o script e o shell que estiver executando o script serão perdidos e serão substituídos pelo novo comando.

Para testar isso em uma janela de shell, experimente executar `exec cat`. Depois que você

teclar Ctrl-D ou Ctrl-C para encerrar o programa `cat`, sua janela deverá desaparecer porque seu processo-filho não existirá mais.

11.11 Subshells

Suponha que você precise alterar levemente o ambiente de um shell, porém não queira que essa mudança seja permanente. Você pode alterar e restaurar uma parte do ambiente (por exemplo, o path ou o diretório de trabalho) usando variáveis de shell, porém é uma maneira desajeitada de trabalhar. A maneira fácil de contornar esses tipos de problema é usar um *subshell* – um processo totalmente novo de shell que pode ser criado somente para executar um ou dois comandos. O novo shell tem uma cópia do ambiente do shell original, e quando o novo shell encerrar, qualquer alteração que tenha sido feita em seu ambiente de shell desaparecerá, deixando o shell inicial executando normalmente.

Para usar um subshell, coloque os comandos a serem executados pelo subshell entre parênteses. Por exemplo, a linha a seguir executa o comando `uglyprogram` em *uglydir* e deixa o shell original intacto:

```
$ (cd uglydir; uglyprogram)
```

Esse exemplo mostra como adicionar um componente ao path, que poderá causar problemas se a alteração for permanente:

```
$ (PATH=/usr/confusing:$PATH; uglyprogram)
```

Usar um subshell para fazer uma alteração a ser usada uma única vez em uma variável de ambiente é uma tarefa tão comum que há até mesmo uma sintaxe pronta que evita o subshell:

```
$ PATH=/usr/confusing:$PATH uglyprogram
```

Pipes e processos em background funcionam também com subshells. O exemplo a seguir usa `tar` para arquivar toda a árvore de diretório em *orig* e, em seguida, descompacta o arquivo no novo diretório *target*, que duplica os arquivos e as pastas em *orig* (isso é útil, pois preserva as informações de propriedade e de permissões, além de ser geralmente mais rápido que usar um comando como `cp -r`):

```
$ tar cf - orig | (cd target; tar xvf -)
```

 **Aviso:** confira muito bem esse tipo de comando antes de executá-lo para garantir que o diretório *target* exista e seja totalmente diferente do diretório *orig*.

11.12 Incluindo outros arquivos em scripts

Se você precisar incluir outro arquivo em seu shell script, utilize o operador ponto (`.`).

Por exemplo, a linha a seguir executa os comandos que estão no arquivo *config.sh*:

```
. config.sh
```

Essa sintaxe para “inclusão” de arquivo não inicia um subshell, e pode ser útil para um grupo de scripts que deva usar um único arquivo de configuração.

11.13 Lendo dados de entrada do usuário

O comando `read` lê uma linha de texto da entrada-padrão e armazena esse texto em uma variável. Por exemplo, o comando a seguir armazena a entrada em `$var`:

```
$ read var
```

É um comando pronto do shell, que pode ser útil em conjunto com outros recursos do shell que não foram mencionados neste livro.

11.14 Quando (não) usar shell scripts

O shell é tão rico em recursos que é difícil condensar seus elementos importantes em um único capítulo. Se você estiver interessado em outras funcionalidades do shell, dê uma olhada em alguns dos livros sobre programação de shell, como *Unix Shell Programming*, 3ª edição, de Stephen G. Kochan e Patrick Wood (SAMS Publishing, 2003), ou na discussão sobre shell scripts no livro *The UNIX Programming Environment* de Brian W. Kernighan e Rob Pike (Prentice Hall, 1984).

Entretanto, em algum momento (especialmente quando você começar a usar o comando `read`), será preciso perguntar a si mesmo se você continua usando a ferramenta certa para o trabalho. Lembre-se daquilo que os shell scripts fazem de melhor: manipulam arquivos e comandos simples. Como afirmamos anteriormente, se você se vir escrevendo algo que pareça confuso, especialmente se envolver operações com strings ou operações aritméticas complicadas, provavelmente você deverá dar uma olhada em uma linguagem de scripting como Python, Perl ou awk.

CAPÍTULO 12

Movendo arquivos pela rede

Este capítulo analisa opções para transferir e compartilhar arquivos entre computadores em uma rede. Começaremos dando uma olhada em algumas maneiras de copiar arquivos que não sejam por meio dos utilitários `scp` e `sftp` que já vimos. Em seguida, veremos brevemente o verdadeiro compartilhamento de arquivos, em que um diretório em um computador é associado a outro computador.

Este capítulo descreve algumas maneiras alternativas de transferir arquivos, pois nem todos os problemas de transferência de arquivos são iguais. Às vezes, é preciso prover um acesso rápido temporário a computadores sobre os quais você não sabe muito; em outras, é preciso manter cópias de estruturas de diretório grandes de modo eficiente e, às vezes, você precisará de acessos mais constantes.

12.1 Cópia rápida

Vamos supor que você queira copiar um arquivo (ou arquivos) de seu computador para outro em sua rede, e que você não se preocupe em copiá-lo de volta nem precise fazer nada muito sofisticado. Você só quer fazer isso rapidamente. Há uma maneira conveniente de fazer isso com Python. Basta acessar o diretório contendo o(s) arquivo(s) e executar:

```
$ python -m SimpleHTTPServer
```

Esse comando inicia um servidor web básico que torna o diretório disponível a qualquer navegador na rede. Normalmente, ele executa na porta 8000, portanto, se o computador em que você executar esse comando estiver em 10.1.2.4, acesse `http://10.1.2.4:8000` no destino e você poderá obter o que precisar.

12.2 rsync

Se quiser mover toda uma estrutura de diretórios por aí, você poderá fazer isso usando `scp -r` — ou, se precisar de um desempenho um pouco melhor, poderá utilizar `tar` em um pipeline:

```
$ tar cBvf - diretório | ssh host_remoto tar xBvpf -
```

Esses métodos fazem o trabalho, porém não são muito flexíveis. Em particular, depois que a transferência terminar, o host remoto poderá não ter uma cópia exata do diretório. Se *diretório* já existir no computador remoto e contiver alguns arquivos extras, esses arquivos serão mantidos após a transferência.

Se precisar realizar esse tipo de tarefa regularmente (em especial, se você planeja automatizar o processo), utilize um sistema sincronizador dedicado. No Linux, o `rsync` é o sincronizador-padrão e oferece um bom desempenho, além de disponibilizar várias maneiras de realizar transferências. Discutiremos alguns dos modos de operação essenciais do `rsync` e veremos algumas de suas peculiaridades.

12.2.1 Básico sobre o `rsync`

Para fazer o `rsync` funcionar entre dois hosts, o programa `rsync` deve ser instalado tanto na origem quanto no destino, e será preciso ter uma maneira de acessar um computador a partir do outro. A maneira mais simples de transferir arquivos é usar uma conta remota de shell, e iremos supor que você vai querer transferir arquivos usando acesso SSH. Entretanto lembre-se de que o `rsync` pode ser prático até mesmo para copiar arquivos e diretórios entre diferentes locais em um único computador, por exemplo, de um sistema de arquivos para outro.

Em sua superfície, o comando `rsync` não é muito diferente de `scp`. Com efeito, `rsync` pode ser executado com os mesmos argumentos. Por exemplo, para copiar um grupo de arquivos para o seu diretório home em *host*, digite:

```
$ rsync arquivo1 arquivo2 ... host:
```

Em qualquer sistema moderno, `rsync` supõe que você estará usando SSH para se conectar a um host remoto.

Tome cuidado com a mensagem de erro a seguir:

```
rsync not found
rsync: connection unexpectedly closed (0 bytes read so far)
rsync error: error in rsync protocol data stream (code 12) at io.c(165)
```

Esse aviso diz que seu shell remoto não conseguiu encontrar o `rsync` em seu sistema. Se `rsync` não estiver no path remoto, mas estiver no sistema, utilize `--rsync-path=path` para especificar manualmente a sua localização.

Se seu nome de usuário for diferente no host remoto, adicione *usuário@* ao nome do host, em que *usuário* é seu nome de usuário em *host*:

```
$ rsync arquivo1 arquivo2 ... usuário@host:
```

A menos que você forneça opções extras, `rsync` copiará somente arquivos. Com efeito, se

você especificar somente as opções descritas até agora e fornecer um diretório *dir* como argumento, você verá a mensagem a seguir:

```
skipping directory dir
```

Para transferir hierarquias inteiras de diretórios – de forma completa, com links simbólicos, permissões, modos e dispositivos –, utilize a opção *-a*. Além do mais, se quiser fazer a cópia para outro lugar que não seja o seu diretório *home* no host remoto, coloque esse destino após o host remoto, da seguinte maneira:

```
$ rsync -a dir host:dir_destino
```

Copiar diretórios pode ser problemático, portanto, se você não estiver totalmente certo do que acontecerá quando transferir os arquivos, utilize a combinação de opções *-nv*. A opção *-n* diz ao *rsync* para operar em modo “dry run”, ou seja, para executar uma tentativa sem realmente copiar qualquer arquivo. A opção *-v* corresponde ao modo *verbose* (extenso), que mostra detalhes sobre a transferência e os arquivos envolvidos:

```
$ rsync -nva dir host:dir_destino
```

A saída terá a aparência a seguir:

```
building file list ... done
ml/nftrans/nftrans.html
[more files]
wrote 2183 bytes read 24 bytes 401.27 bytes/sec
```

12.2.2 Criando cópias exatas de uma estrutura de diretório

Por padrão, o *rsync* copia arquivos e diretórios sem considerar o conteúdo anterior do diretório de destino. Por exemplo, se você transferir o diretório *d* contendo os arquivos *a* e *b* para um computador que já tenha um arquivo chamado *d/c*, o destino conterà *d/a*, *d/b* e *d/c* depois do *rsync*.

Para criar uma réplica exata do diretório de origem, você deve apagar os arquivos no diretório de destino que não existam no diretório de origem, como *d/c* nesse exemplo. Utilize a opção *--delete* para isso:

```
$ rsync -a --delete dir host:dir_destino
```

 **AVISO:** isso pode ser perigoso, pois, normalmente, você deve inspecionar o diretório de destino para ver se não há algo que você possa apagar inadvertidamente. Lembre-se de que, se você não estiver certo sobre a sua transferência, a opção *-n* poderá ser usada para executar um *dry run* e saber exatamente quando o *rsync* vai querer apagar um arquivo.

12.2.3 Usando a barra no final

Tome cuidado particularmente ao especificar um diretório como origem em uma linha

de comando `rsync`. Considere o comando básico com o qual trabalhamos até agora:

```
$ rsync -a dir host:dir_dest
```

Quando o comando terminar, você terá um diretório *dir* dentro de *dir_dest* em *host*. A figura 12.1 mostra um exemplo de como `rsync` normalmente trata um diretório contendo arquivos de nomes *a* e *b*. Entretanto adicionar uma barra (/) altera significativamente o comportamento:

```
$ rsync -a dir/ host:dir_dest
```

Nesse caso, `rsync` copia tudo o que estiver *em dir* para *dir_dest* em *host*, sem realmente criar *dir* no host destino. Desse modo, podemos pensar na transferência de *dir/* como uma operação semelhante a `cp dir/* dir_dest` no sistema de arquivos local.

Por exemplo, suponha que você tenha um diretório *dir* contendo os arquivos *a* e *b* (*dir/a* e *dir/b*). Execute a versão do comando com barra no final para transferir os arquivos para o diretório *dir_dest* em *host*:

```
$ rsync -a dir/ host:dir_dest
```

Quando a transferência for concluída, *dir_dest* conterá cópias de *a* e de *b*, mas *não* de *dir*. Porém, se você omitir a / final em *dir*, *dir_dest* terá uma cópia de *dir*, com *a* e *b* dentro dele. Então, como resultado da transferência, você teria arquivos e diretórios de nomes *dir_dest/dir/a* e *dir_dest/dir/b* no host remoto. A figura 12.2 mostra como `rsync` lida com a estrutura de diretório da figura 12.1 quando uma barra no final estiver sendo usada.

Ao transferir arquivos e diretórios para um host remoto, adicionar uma / acidentalmente depois de um path normalmente não representará nada além de um incômodo; você poderia acessar o *host* remoto, adicionar o diretório *dir* e colocar todos os itens transferidos de volta em *dir*. Infelizmente, você deve tomar cuidado para evitar desastres ao combinar a / final com a opção `--delete`, pois dessa maneira, você poderá facilmente remover arquivos que você não tinha a intenção de apagar.

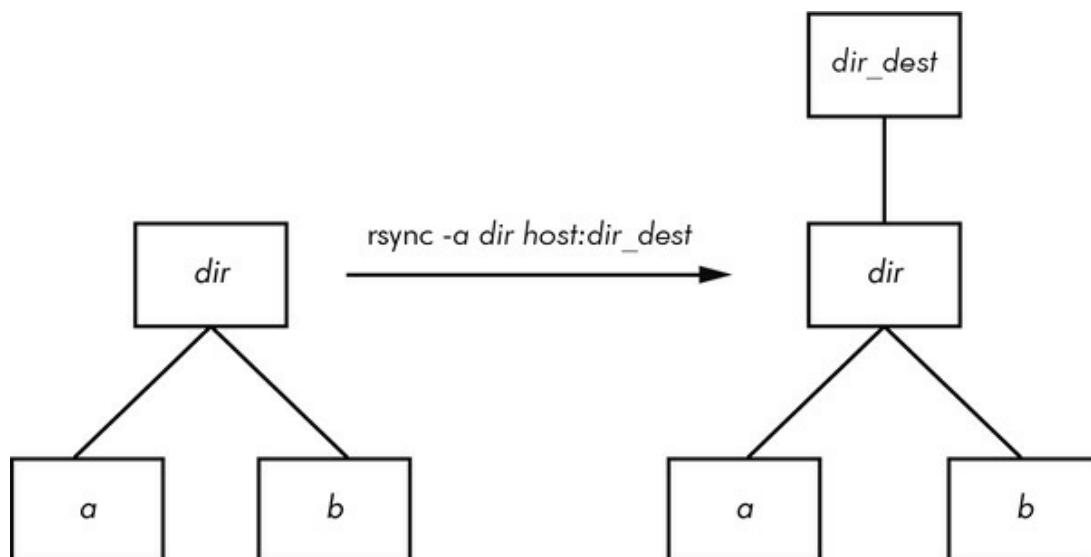


Figura 12.1 – Cópia normal com rsync.

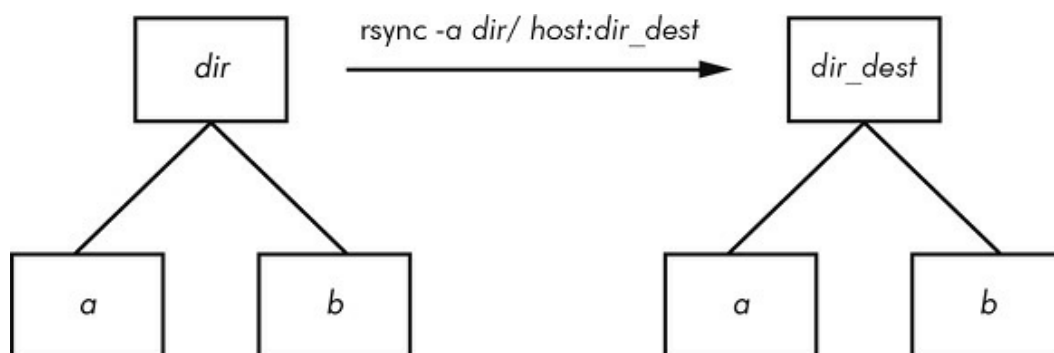


Figura 12.2 – Efeito da barra final em rsync.

👍 **Observação:** tome cuidado com o recurso de preenchimento automático de nome de arquivo de seu shell. O readline do GNU e várias outras bibliotecas de preenchimento colocam barras finais nos nomes de diretório que tenham sido preenchidos.

12.2.4 Excluindo arquivos e diretórios

Um recurso bastante importante de `rsync` é sua capacidade de excluir arquivos e diretórios de uma operação de transferência. Por exemplo, suponha que você queira transferir um diretório local chamado *src* para *host*, mas queira excluir tudo o que se chamar *.git*. Isso pode ser feito da seguinte maneira:

```
$ rsync -a --exclude=.git src host:
```

Observe que esse comando exclui *todos* os arquivos e diretórios de nome *.git* porque `--exclude` recebe um padrão, e não um nome de arquivo absoluto. Para excluir um item específico, defina um path absoluto que comece com `/`, como mostrado a seguir:

```
$ rsync -a --exclude=/src/.git src host:
```

👍 **Observação:** a primeira `/` em `/src/.git` nesse comando não é o diretório-raiz de seu sistema, mas o

diretório-base da transferência.

A seguir, apresentamos mais algumas dicas sobre como excluir padrões:

- Você pode ter quantos parâmetros `--exclude` você quiser.
- Se os mesmos padrões forem usados repetidamente, coloque-os em um arquivo em formato texto simples (um padrão por linha) e utilize `--exclude-from=arquivo`.
- Para excluir diretórios de nome *item*, porém incluir arquivos com esse nome, use uma barra no final: `--exclude=item/`.
- O padrão para exclusão é baseado em um componente de nome completo de arquivo ou de diretório e pode conter globs simples (caracteres-coringa). Por exemplo, `t*s` corresponde a *this*, mas não a *ethers*.
- Se você excluir um diretório ou um nome de arquivo, mas achar que seu padrão é restritivo demais, utilize `--include` para incluir especificamente outro arquivo ou diretório.

12.2.5 Integridade na transferência, medidas de segurança e modos verbose

Para agilizar a operação, `rsync` usa uma verificação rápida para determinar se algum arquivo da origem da transferência já está no destino. A verificação rápida utiliza uma combinação do tamanho do arquivo com a data de sua última modificação. Na primeira vez que você transferir uma hierarquia completa de diretório para um host remoto, o `rsync` perceberá que nenhum dos arquivos existe no destino e transferirá tudo. Testar sua transferência com `rsync -n` fará essa verificação para você.

Após ter executado `rsync` uma vez, execute-o novamente usando `rsync -v`. Dessa vez, você verá que nenhum arquivo aparecerá na lista de transferência porque o conjunto de arquivos já existe nos dois lados, com as mesmas datas de modificação.

Quando os arquivos do lado origem não forem idênticos aos arquivos do lado destino, o `rsync` transferirá os arquivos da origem e sobrescreverá qualquer arquivo que existir no lado remoto. No entanto o comportamento-padrão poderá ser inadequado, pois você poderá precisar de garantias adicionais de que os arquivos são realmente os mesmos antes de ignorá-los nas transferências, ou poderá acrescentar algumas medidas de segurança. A seguir, apresentamos algumas opções que podem ser práticas:

- `--checksum` (abreviatura: `-c`) – calcula checksums (assinaturas únicas, na maior parte das vezes) dos arquivos para ver se eles são iguais. Isso consome recursos adicionais de I/O e de CPU durante as transferências, porém, se você estiver

lidando com dados sensíveis ou com arquivos que geralmente tenham tamanhos uniformes, essa opção será obrigatória.

- `--ignore-existing` – não sobrescreve arquivos que já estejam no lado destino.
- `--backup` (abreviatura: `-b`) – não sobrescreve arquivos que já estejam no destino, mas renomeia esses arquivos existentes adicionando-lhes um sufixo `~` aos seus nomes antes de transferir arquivos novos.
- `--suffix=s` – muda o sufixo usado em `--backup` de `~` para `s`.
- `--update` (abreviatura: `-u`) – não sobrescreve nenhum arquivo no destino que tenha uma data posterior à data do arquivo correspondente na origem.

Sem opções especiais, `rsync` funcionará silenciosamente, produzindo uma saída somente quando houver algum problema. Entretanto podemos usar `rsync -v` para ativar o modo verbose ou `rsync -vv` para obter mais detalhes ainda. (Podemos acrescentar quantas opções `v` quisermos, porém duas provavelmente são mais do que o necessário.) Para obter um resumo completo após a transferência, utilize `rsync --stats`.

12.2.6 Compressão

Muitos usuários gostam da opção `-z` em conjunto com `-a` para compactar os dados antes da transmissão:

```
$ rsync -az dir host:dir_destino
```

A compressão pode melhorar o desempenho em determinadas situações, por exemplo, quando uma grande quantidade de dados estiver sendo carregada por meio de uma conexão lenta (como o link lento de upstream em várias conexões DSL) ou quando a latência entre dois hosts for alta. No entanto, em uma rede local rápida, os computadores nas duas extremidades poderão ser limitados pelo tempo de CPU necessário para compactar e descompactar os dados, de modo que uma transferência sem compressão poderá ser mais rápida.

12.2.7 Limitando a largura de banda

É fácil congestionar o uplink de conexões de Internet ao carregar uma grande quantidade de dados para um host remoto. Mesmo que você não vá usar sua capacidade de downlink (geralmente alta) durante uma transferência como essa, sua conexão continuará a parecer bem lenta se você permitir que `rsync` funcione da forma mais rápida que puder, pois os pacotes TCP de saída, como as solicitações HTTP, terão de competir com suas transferências pela largura de banda em seu uplink.

Para contornar esse problema, utilize `--bwlimit` para dar um pouco de espaço para seu

uplink respirar. Por exemplo, para limitar a largura de banda para 10.000 Kpbs, você poderá executar algo como:

```
$ rsync --bwlimit=10000 -a dir host:dir_destino
```

12.2.8 Transferindo arquivos para o seu computador

O comando `rsync` não serve somente para copiar arquivos de seu computador local para um host remoto. Também é possível transferir arquivos de um computador remoto para o seu host local ao especificar o host remoto e o path de origem remoto como primeiro argumento na linha de comando. Desse modo, para transferir *dir_orig* do host para *dir_dest* no host local, execute o comando a seguir:

```
$ rsync -a host:dir_orig dir_dest
```

 **Observação:** como mencionamos anteriormente, podemos usar `rsync` para duplicar diretórios em seus computadores locais se *host:* for totalmente omitido.

12.2.9 Tópicos adicionais associados ao rsync

Sempre que você precisar copiar vários arquivos, `rsync` deverá ser um dos primeiros utilitários a surgir em sua mente. Executar `rsync` em modo batch (lote) é particularmente útil, e você encontrará diversas opções para empregar arquivos auxiliares relacionados a opções do comando, logging e estado de transferência. Em particular, os arquivos de estado tornam as transferências longas mais rápidas e mais fáceis de serem retomadas quando interrompidas.

Você também achará `rsync` útil para criar backups. Por exemplo, podemos associar áreas de armazenamento na Internet, como o S3 da Amazon, ao seu sistema Linux e então usar `rsync --delete` para sincronizar periodicamente um sistema de arquivos com a área de armazenamento na rede a fim de criar um sistema de backup bastante eficiente.

Há muitas outras opções de linha de comando além daquelas descritas aqui. Para obter uma visão geral deles, execute `rsync --help`. Você encontrará informações mais detalhadas na página de manual `rsync(1)`, assim como na página inicial de `rsync` em <http://rsync.samba.org/>.

12.3 Introdução ao compartilhamento de arquivos

Seu computador Linux provavelmente não está sozinho em sua rede, e quando há vários computadores em uma rede, quase sempre há um motivo para compartilhar arquivos entre eles. No restante deste capítulo, estaremos preocupados principalmente com o compartilhamento de arquivos entre computadores Windows e Mac OS X porque é

interessante ver como o Linux se adapta totalmente a ambientes diferentes. Visando a compartilhar arquivos entre computadores Linux ou acessar arquivos de um dispositivo NAS (Network Area Storage), discutiremos brevemente o uso do NFS (Network File System) como um cliente.

12.4 Compartilhando arquivos com o Samba

Se você tiver computadores executando Windows, provavelmente vai querer permitir acesso aos arquivos e às impressoras de seu sistema Linux a partir desses computadores Windows usando o SMB (Server Message Block, ou Bloco de mensagens de servidor), que é o protocolo-padrão de rede do Windows. O Mac OS X também suporta compartilhamento de arquivos com SMB.

O pacote-padrão de software para compartilhamento de arquivos no Unix chama-se Samba. O Samba não só permite que os computadores Windows de sua rede acessem seu sistema Linux como também funciona ao contrário: você pode imprimir e acessar arquivos nos servidores Windows a partir de seu computador Linux com o software cliente do Samba.

Para configurar um servidor Samba, execute os passos a seguir:

1. Crie um arquivo *smb.conf*.
2. Adicione seções para compartilhamento de arquivos em *smb.conf*.
3. Adicione seções para compartilhamento de impressoras em *smb.conf*.
4. Inicie os daemons *nmbd* e *smbd* do Samba.

Ao instalar o Samba a partir do pacote de uma distribuição, seu sistema deverá executar os passos listados anteriormente usando alguns defaults razoáveis para o servidor. Entretanto é provável que ele não vá ser capaz de determinar quais *compartilhamentos* (recursos) em particular serão oferecidos aos clientes em seu computador Linux.

 **Observação:** a discussão sobre o Samba neste capítulo será breve e estará limitada a fazer com que computadores Windows em uma única sub-rede vejam um computador Linux independente por meio do navegador Windows Network Places. Há inúmeras maneiras de configurar o Samba, pois há várias possibilidades para controle de acesso e topologia de rede. Para obter detalhes mais minuciosos sobre como configurar um servidor de larga escala, dê uma olhada no livro *Using Samba*, 3ª edição (O'Reilly, 2007), que é um guia muito mais completo, e acesse o site do Samba em <http://www.samba.org/>.

12.4.1 Configurando o servidor

O arquivo principal de configuração do Samba é *smb.conf*, que a maioria das

distribuições coloca em um diretório de *etc*, por exemplo, em */etc/samba*. No entanto pode ser que você tenha de procurar esse arquivo, e ele também poderá estar em um diretório *lib* como */usr/local/samba/lib*.

O arquivo *smb.conf* é semelhante ao estilo XDG que já vimos em outros lugares (como no formato da configuração do *systemd*) e está dividido em várias seções indicadas por colchetes (como *[global]* e *[printers]*). A seção *[global]* em *smb.conf* contém opções gerais que se aplicam a todo o servidor e a todos os compartilhamentos. Essas opções dizem respeito principalmente à configuração de rede e ao controle de acesso. A seção *[global]* de exemplo a seguir mostra como definir o nome do servidor, a descrição e o grupo de trabalho:

```
[global]
# nome do servidor
netbios name = name
# descrição do servidor
server string = My server via Samba
# grupo de trabalho (workgroup)
workgroup = MYNETWORK
```

Esses parâmetros funcionam do seguinte modo:

- *netbios name* — é o nome do servidor. Se esse parâmetro for omitido, o Samba usará o nome de host Unix.
- *server string* — é uma breve descrição do servidor. O default é o número de versão do Samba.
- *workgroup* — é o nome do grupo de trabalho SMB. Se você estiver em um domínio Windows, configure esse parâmetro com o nome de seu domínio.

12.4.2 Controle de acesso ao servidor

Você pode adicionar opções em seu arquivo *smb.conf* para limitar os computadores e os usuários que podem acessar seu servidor Samba. A lista a seguir inclui várias opções que podem ser definidas em sua seção *[global]* e nas seções que controlam os compartilhamentos individuais (conforme será descrito mais adiante neste capítulo):

- *interfaces* — configure essa opção para fazer o Samba ouvir as redes ou as interfaces especificadas. Por exemplo:

```
interfaces = 10.23.2.0/255.255.255.0
interfaces = eth0
```

- *bind interfaces only* — configure essa opção com *yes* ao usar o parâmetro *interfaces* a fim de limitar o acesso aos computadores que podem ser alcançados por meio dessas

interfaces.

- `valid users` — configure essa opção para conceder acesso aos usuários especificados. Por exemplo:

```
valid users = jruser, bill
```

- `guest ok` — configure esse parâmetro com `true` para tornar um compartilhamento disponível aos usuários anônimos na rede.
- `guest only` — configure esse parâmetro com `true` para permitir somente acesso anônimo.
- `browseable` — configure esse parâmetro para tornar os compartilhamentos visíveis aos navegadores de rede. Se esse parâmetro for configurado com `no` para qualquer compartilhamento, você ainda poderá acessar os compartilhamentos no servidor Samba, porém será preciso conhecer seus nomes exatos para poder acessá-los.

12.4.3 Senhas

Em geral, você deverá permitir acesso ao seu servidor Samba somente mediante autenticação com senha. Infelizmente, o sistema básico de senhas do Unix é diferente do sistema do Windows, portanto, a menos que você especifique senhas de rede em formato texto simples ou as autentique usando um servidor Windows, será preciso configurar um sistema de senhas alternativo. Esta seção mostra como configurar um sistema desse tipo usando o backend TDB (Trivial Database) do Samba, que é apropriado para redes pequenas.

Inicialmente, use as entradas a seguir em sua seção `[global]` de *smb.conf* para definir as características do banco de dados de senhas do Samba:

```
# usa o tdb para Samba para permitir senhas criptografadas
security = user
passdb backend = tdbsam
obey pam restrictions = yes
smb passwd file = /etc/samba/passwd_smb
```

Essas linhas permitem manipular o banco de dados de senhas do Samba usando o comando `smbpasswd`. O parâmetro `obey pam restrictions` garante que qualquer usuário que alterar sua senha usando o comando `smbpasswd` deverá obedecer a qualquer regra imposta pelo PAM para alterações normais de senha. Para o parâmetro `passdb backend`, você poderá adicionar um nome de path opcional para o arquivo TDB após dois-pontos; por exemplo, `tdbsam:/etc/samba/private/passwd.tdb`.

 **Observação:** se você tiver acesso a um domínio Windows, será possível definir `security = domain` para fazer o Samba usar os nomes de usuário do domínio e acabar com a necessidade de ter um banco de dados de senhas. Entretanto, para que os usuários do domínio possam acessar o computador executando Samba, cada

usuário do domínio deverá ter uma conta local com o mesmo nome de usuário nesse computador.

Adicionando e apagando usuários

A primeira tarefa que você deve fazer para conceder acesso ao seu servidor Samba a um usuário Windows é adicionar o usuário ao banco de dados de senhas usando o comando `smbpasswd -a`:

```
# smbpasswd -a nome_do_usuario
```

O parâmetro *nome_do_usuario* no comando `smbpasswd` deve ser um nome de usuário válido em seu sistema Linux.

Assim como o programa `passwd` normal do sistema, `smbpasswd` solicitará a senha do novo usuário duas vezes. Se a senha passar por todas as verificações de segurança necessárias, `smbpasswd` confirmará que um novo usuário foi criado.

Para remover um usuário, utilize a opção `-x` de `smbpasswd`:

```
# smbpasswd -x nome_do_usuario
```

Para desativar temporariamente um usuário, utilize a opção `-d`; a opção `-e` irá reativá-lo:

```
# smbpasswd -d nome_do_usuario
```

```
# smbpasswd -e nome_do_usuario
```

Alterando senhas

Você pode alterar uma senha do Samba como superusuário usando `smbpasswd` sem opções ou palavras-chave além do nome do usuário:

```
# smbpasswd nome_do_usuario
```

No entanto, se o servidor Samba estiver executando, qualquer usuário poderá mudar sua própria senha no Samba usando `smbpasswd` sozinho na linha de comando.

Por fim, eis um local de sua configuração em que você deverá prestar atenção. Se você vir uma linha como a que se segue em seu arquivo *smb.conf*, tome cuidado:

```
unix password sync = yes
```

Essa linha faz o `smbpasswd` mudar a senha normal de um usuário, *além* de mudar a senha do Samba. O resultado pode ser bastante confuso, especialmente quando um usuário mudar sua senha do Samba para algo que seja diferente de sua senha Linux e descobrir que não poderá mais fazer login. Algumas distribuições definem esse parâmetro por padrão em seus pacotes de servidores Samba!

12.4.4 Iniciando o servidor

Poderá ser necessário iniciar o seu servidor caso você não tenha instalado o Samba a

partir de um pacote da distribuição. Para isso, execute `nmbd` e `smbd` com os argumentos a seguir, em que `arq_config_smb` é o path completo de seu arquivo `smb.conf`:

```
# nmbd -D -s arq_config_smb
# smbd -D -s arq_config_smb
```

O daemon `nmbd` é um servidor de nomes NetBIOS, e `smbd` faz o verdadeiro trabalho de lidar com as solicitações de compartilhamento. A opção `-D` especifica o modo daemon. Se o arquivo `smb.conf` for alterado enquanto `smbd` estiver executando, você poderá notificar o daemon a respeito das mudanças usando um sinal HUP ou pode utilizar o comando de reinicialização de serviço de sua distribuição (por exemplo, `systemctl` ou `initctl`).

12.4.5 Diagnósticos e arquivos de log

Se algo der errado ao iniciar um dos servidores Samba, uma mensagem de erro aparecerá na linha de comando. No entanto as mensagens de diagnóstico de tempo de execução serão enviadas para os arquivos de log `log.nmbd` e `log.smbd`, que normalmente estão em um diretório de `/var/log`, por exemplo, `/var/log/samba`. Outros arquivos de log também poderão ser encontrados nesse local, como os logs individuais de cada cliente em particular.

12.4.6 Configurando um compartilhamento de arquivo

Para exportar um diretório a clientes SMB (ou seja, para compartilhar um diretório com um cliente), adicione uma seção como a que se segue ao seu arquivo `smb.conf`, em que *label* é o nome que você quer dar ao compartilhamento e *path* é o path completo do diretório:

```
[label]
path = path
comment = descrição do compartilhamento
guest ok = no
writable = yes
printable = no
```

Os parâmetros a seguir são úteis nos compartilhamentos de diretório:

- `guest ok` – permite acesso de convidados (`guest`) ao compartilhamento. O parâmetro `public` é um sinônimo.
- `writable` – Uma configuração igual a `yes` ou `true` nesse local marca o compartilhamento como leitura-escrita. Não permita acesso de convidados em um compartilhamento de leitura-escrita.

- `printable` — especifica um compartilhamento de impressora. Esse parâmetro deve ser configurado com `no` ou com `false` para um compartilhamento de diretório.
- `veto files` — impede a exportação de qualquer arquivo que corresponda aos padrões especificados. Cada padrão deve ser colocado entre barras (de modo que se pareça com `/padrão/`). O exemplo a seguir exclui arquivos objeto bem como qualquer arquivo ou diretório chamado *bin*:

```
veto files = /*.o/bin/
```

12.4.7 Diretórios home

Uma seção `[homes]` pode ser acrescentada ao seu arquivo *smb.conf* se você quiser exportar diretórios home aos usuários. A seção deverá ter um aspecto semelhante a:

```
[homes]
comment = home directories
browseable = no
writable = yes
```

Por padrão, o Samba lê a entrada de */etc/passwd* dos usuários logados para determinar o diretório home para `[homes]`. Porém, se você não quiser que o Samba tenha esse comportamento (ou seja, você quer manter os diretórios home do Windows em um local diferente dos diretórios home normais do Linux), a substituição `%S` poderá ser usada em um parâmetro `path`. Por exemplo, eis o modo como você pode mudar o diretório `[homes]` de um usuário para */u/user*:

```
path = /u/%S
```

O Samba substitui `%S` pelo nome do usuário corrente.

12.4.8 Compartilhando impressoras

Você pode exportar todas as suas impressoras para clientes Windows ao adicionar uma seção `[printers]` ao seu arquivo *smb.conf*. A seguir, apresentamos a aparência dessa seção quando você estiver usando CUPS — o sistema-padrão de impressão do Unix:

```
[printers]
comment = Printers
browseable = yes
printing = CUPS
path = cups
printable = yes
writable = no
```

Para usar o parâmetro `printing = CUPS`, sua instalação do Samba deve estar configurada e ligada à biblioteca CUPS.

👍 **Observação:** de acordo com a sua configuração, você também poderá querer permitir acessos de convidados às suas impressoras usando a opção `guest ok = yes` em vez de dar uma senha ou conta do Samba a todos que precisarem ter acesso às impressoras. Por exemplo, é fácil limitar o acesso às impressoras em uma única sub-rede usando regras de firewall.

12.4.9 Usando o cliente do Samba

O programa cliente do Samba – `smbclient` – pode imprimir e acessar compartilhamentos remotos do Windows. Esse programa é prático se você estiver em um ambiente em que será preciso interagir com servidores Windows, que não oferecem um meio de comunicação amigável com o Unix.

Para começar a trabalhar com o `smbclient`, utilize a opção `-L` para obter uma lista de compartilhamentos de um servidor remoto chamado *SERVIDOR*:

```
$ smbclient -L -U nome_do_usuario SERVIDOR
```

`-U nome_do_usuario` não será necessário se seu nome de usuário Linux for igual ao seu nome de usuário em *SERVIDOR*.

Após executar esse comando, `smbclient` pedirá uma senha. Para tentar acessar um compartilhamento como convidado, tecele Enter; caso contrário, forneça a sua senha em *SERVIDOR*. Se houver sucesso, você deverá obter uma lista de compartilhamentos como esta:

Sharename	Type	Comment
-----	----	-----
Software	Disk	Software distribution
Scratch	Disk	Scratch space
IPC\$	IPC	IPC Service
ADMIN\$	IPC	IPC Service
Printer1	Printer	Printer in room 231A
Printer2	Printer	Printer in basement

Use o campo `Type` para conseguir entender o que é cada compartilhamento e preste atenção somente nos compartilhamentos `Disk` e `Printer` (os compartilhamentos `IPC` servem para gerenciamento remoto). Essa lista tem dois compartilhamentos de disco e dois de impressora. Use o nome na coluna `Sharename` para acessar cada compartilhamento.

12.4.10 Acessando arquivos como cliente

Se você precisar somente de acessos ocasionais aos arquivos em um compartilhamento de disco, utilize o comando a seguir. (Novamente, você pode omitir `-U nome_do_usuario` se o seu nome de usuário Linux corresponder ao seu nome de usuário no servidor.)

```
$ smbclient -U nome_do_usuario '\\SERVIDOR\nome_do_compartilhamento'
```

Se houver sucesso, você obterá um prompt como o que se segue, indicando que você poderá agora transferir arquivos:

```
smb: \>
```

Nesse modo de transferência de arquivos, `smbclient` é semelhante ao `ftp` do Unix, e os comandos a seguir poderão ser executados:

- `get arquivo` – copia *arquivo* do servidor remoto para o diretório corrente local.
- `put arquivo` – copia *arquivo* do computador local para o servidor remoto.
- `cd dir` – muda o diretório no servidor remoto para *dir*.
- `lcd dirlocal` – muda o diretório corrente local para *dirlocal*.
- `pwd` – exibe o diretório corrente no servidor remoto, incluindo os nomes do servidor e do compartilhamento.
- `!comando` – executa *comando* no host local. Dois comandos particularmente práticos são `!pwd` e `!ls` para determinar o diretório e o status de arquivos do lado local.
- `help` – mostra uma lista completa dos comandos.

Usando o sistema de arquivos CIFS

Se você precisar de acessos frequentes e regulares a arquivos em um servidor Windows, será possível associar um compartilhamento diretamente ao seu sistema usando `mount`. A sintaxe do comando está sendo mostrada a seguir. Observe o uso de *SERVIDOR:nome_do_compartilhamento* em vez do formato `\\SERVIDOR\ nome_do_compartilhamento` normal.

```
# mount -t cifs SERVIDOR:nome_do_compartilhamento ponto_de_montagem -o  
user=nome_do_usuario,pass=senha
```

Para usar `mount` dessa maneira, você deve ter os utilitários CIFS (Common Internet File System) disponíveis para o Samba. A maioria das distribuições os disponibiliza como um pacote separado.

12.5 Clientes NFS

O sistema-padrão para compartilhamento de arquivos nos sistemas Unix é o NFS; há várias versões diferentes do NFS para diferentes cenários. Você pode disponibilizar o NFS sobre TCP e UDP, com uma grande quantidade de técnicas de autenticação e de criptografia. Pelo fato de haver tantas opções, o NFS pode ser um assunto amplo, portanto vamos nos ater ao básico sobre os clientes NFS.

Para montar um diretório remoto em um servidor com NFS, utilize a mesma sintaxe

básica usada para montar um diretório CIFS:

```
# mount -t nfs servidor:diretório ponto_de_montagem
```

Tecnicamente, a opção `-t nfs` não é necessária, pois `mount` deverá descobrir isso, mas você pode investigar as opções na página de manual `nfs(5)`. (Você encontrará várias opções diferentes para segurança usando a opção `sec`. Muitos administradores de redes pequenas e fechadas usam controle de acesso baseado em `host`. No entanto, métodos mais sofisticados como a autenticação baseada em Kerberos exigem configurações adicionais em outras partes de seu sistema.)

Quando perceber que você está fazendo mais uso de sistemas de arquivos por uma rede, configure o sistema de montagem automática para que seu sistema monte os sistemas de arquivos somente quando você realmente tentar usá-los para evitar problemas com dependências no boot. A ferramenta tradicional de montagem automática chama-se `automount`, e há uma versão mais nova chamada `amd`, porém boa parte disso atualmente está sendo suplantada pelo tipo de unidade `automount` do `systemd`.

12.6 Outras opções e limitações do serviço de arquivos em redes

Configurar um servidor NFS para compartilhar arquivos com outros computadores Linux é mais complicado que usar um cliente NFS simples. É preciso executar os daemons do servidor (`mountd` e `nfsd`) e configurar o arquivo `/etc/exports` para que reflita os diretórios que você estiver compartilhando. No entanto não discutiremos os servidores NFS principalmente porque uma área de armazenamento compartilhada por meio de uma rede geralmente se tornará muito mais conveniente simplesmente com a aquisição de um dispositivo NAS que cuide disso para você. Muitos desses dispositivos são baseados em Linux, portanto terão naturalmente suporte ao servidor NFS. Os fornecedores agregam valor a seus dispositivos NAS ao oferecerem suas próprias ferramentas de administração de modo a eliminar as dificuldades de tarefas tediosas como instalar configurações RAID e backups na nuvem.

Falando em backups na nuvem, outra opção de serviço de arquivo em redes é o armazenamento na nuvem. Isso pode ser prático quando você precisar da área extra de armazenamento para backups automáticos e não se importar que haja um pouco de impacto no desempenho. É especialmente útil quando você não precisar do serviço durante um longo período de tempo ou não precisar acessá-lo com muita frequência. Geralmente, você poderá montar uma área de armazenamento na Internet de modo muito semelhante ao que faria com NFS.

Embora o NFS e outros sistemas de compartilhamento de arquivos funcione bem para usos ocasionais, não espere um ótimo desempenho. Acessos somente de leitura a arquivos maiores devem funcionar bem, por exemplo, quando você estiver fazendo streaming de áudio ou de vídeo, pois os dados estarão sendo lidos em porções grandes e previsíveis, que não exigem muita comunicação de um lado para outro entre o servidor de arquivos e seu cliente. Desde que a rede seja rápida o bastante e o cliente tenha memória suficiente, um servidor poderá fornecer os dados conforme for necessário.

O armazenamento local é muito mais rápido para tarefas que envolvam vários arquivos pequenos, como compilar pacotes de software e iniciar ambientes de desktop. A situação torna-se mais complicada quando você tem uma rede maior, com vários usuários acessando diversos computadores diferentes, pois nesse caso há um compromisso entre conveniência, desempenho e facilidade de administração.

CAPÍTULO 13

Ambientes de usuário

O foco principal deste livro está no sistema Linux, normalmente subjacente aos processos servidores e às sessões interativas de usuário. Porém, em algum momento, o sistema e o usuário deverão se encontrar em algum lugar. Os arquivos de inicialização têm um papel importante nesse ponto, pois eles definem defaults para o shell e para outros programas interativos. Esses arquivos definem como o sistema se comportará quando um usuário fizer login.

A maioria dos usuários não presta muita atenção em seus arquivos de inicialização e entra em contato com eles somente quando quer acrescentar algo por conveniência, por exemplo, um alias. Ao longo do tempo, os arquivos ficam cheios de variáveis de ambiente e testes desnecessários que podem resultar em problemas incômodos (ou bem sérios).

Se já faz algum tempo que você tem seu computador Linux, poderá perceber que seu diretório `home` acumulou um conjunto incrivelmente grande de arquivos de inicialização ao longo do tempo. Às vezes, eles são chamados de arquivos ponto (*dot files*), pois quase sempre começam com um ponto (`.`). Muitos deles são criados automaticamente quando um programa é executado pela primeira vez, e você jamais precisará alterá-los. Este capítulo discute principalmente os arquivos de inicialização de shell, que são os mais prováveis de serem alterados ou reescritos do zero. Inicialmente, vamos dar uma olhada no nível de cuidado que você deve tomar ao trabalhar com esses arquivos.

13.1 Diretrizes para criar arquivos de inicialização

Ao criar arquivos de inicialização, tenha o usuário em mente. Se você for o único usuário em um computador, não haverá muito com que se preocupar, pois os erros afetarão somente você e serão fáceis de serem corrigidos. Entretanto, se você estiver criando arquivos de configuração com o propósito de serem defaults para todos os novos usuários em um computador ou na rede, ou se você achar que alguém poderá copiar seus arquivos para serem usados em um computador diferente, sua tarefa se tornará consideravelmente mais difícil. Se você cometer um erro em um arquivo de

inicialização para dez usuários, poderá acabar tendo de corrigir esse erro dez vezes. Tenha dois objetivos essenciais em mente ao criar arquivos de inicialização para outros usuários:

- Simplicidade – mantenha a quantidade de arquivos de inicialização baixa e faça com que os arquivos sejam tão pequenos e simples quanto possíveis para que sejam fáceis de ser modificados, porém difíceis de apresentar problemas. Cada item em um arquivo de inicialização é somente um item a mais que poderá causar problemas.
- Legibilidade – use comentários extensos nos arquivos para que os usuários tenham uma boa ideia do que cada parte de um arquivo faz.

13.2 Quando os arquivos de inicialização devem ser alterados

Antes de fazer uma alteração em um arquivo de inicialização, pergunte a si mesmo se essa alteração realmente deve ser feita. A seguir, estão alguns bons motivos para alterar arquivos de inicialização:

- Você quer alterar o prompt default.
- É necessário acomodar algum software crítico instalado localmente. (Inicialmente, porém, considere o uso de scripts wrappers.)
- Seus arquivos de inicialização existentes estão com problema.

Se tudo o que estiver em sua distribuição Linux estiver funcionando, tome cuidado. Às vezes, os arquivos de inicialização default interagem com outros arquivos em */etc*.

Apesar disso, é provável que você não estivesse lendo este capítulo se não estivesse interessado em alterar os defaults, portanto vamos analisar o que é importante.

13.3 Elementos do arquivo de inicialização do shell

O que um arquivo de inicialização de shell contém? Alguns elementos podem parecer óbvios, como o path e uma configuração de prompt. Mas o que, exatamente, *deve* estar no path, e qual é a aparência de um prompt razoável? E quanto é demais para ser colocado em um arquivo de inicialização?

As próximas seções discutirão os aspectos essenciais de um arquivo de inicialização de shell – do path de comandos ao prompt e os aliases até a máscara de permissões.

13.3.1 O path de comandos

A parte mais importante de qualquer arquivo de inicialização de shell é o path de comandos. O path deve incluir os diretórios que contêm todas as aplicações de interesse para um usuário normal. O path deve conter, no mínimo, os componentes a seguir, nesta sequência:

```
/usr/local/bin  
/usr/bin  
/bin
```

Essa sequência garante que você poderá sobrescrever os programas default que sejam padrões por variantes locais específicas em */usr/local*.

A maioria das distribuições Linux instala executáveis para praticamente todos os pacotes de software em */usr/bin*. Há diferenças ocasionais como colocar jogos em */usr/games* e aplicações gráficas em um local separado, portanto verifique os defaults de seu sistema antes. E certifique-se de que todo programa de uso geral do sistema esteja disponível por meio de um dos diretórios listados anteriormente. Do contrário, seu sistema provavelmente estará saindo do controle. Não altere o path default no ambiente de seu usuário para acomodar o diretório de instalação de um novo software. Uma maneira simples de acomodar diferentes diretórios de instalação é usar links simbólicos em */usr/local/bin*.

Muitos usuários usam um diretório *bin* próprio para armazenar shell scripts e programas, portanto pode ser que você queira acrescentar o seguinte na frente do path:

```
$HOME/bin
```

 **Observação:** uma convenção mais recente consiste em colocar os binários em *\$HOME/.local/bin*.

Se você estiver interessado em utilitários de sistema (como *traceroute*, *ping* e *lsmod*), acrescente os diretórios *sbin* ao seu path:

```
/usr/local/sbin  
/usr/sbin  
/sbin
```

Adicionando o ponto (.) ao path

Há um componente do path de comandos a ser discutido que é pequeno, porém controverso: o ponto. Colocar um ponto (.) em seu path permite executar programas no diretório corrente sem usar *./* na frente do nome do programa. Isso pode parecer conveniente ao escrever scripts ou compilar programas, porém não é uma boa ideia por dois motivos:

- Pode representar um problema de segurança. *Jamais* coloque um ponto na frente do

path. Eis um exemplo do que pode acontecer: um invasor pode colocar um cavalo de Troia chamado `ls` em um arquivo distribuído pela Internet. Mesmo que o ponto estivesse no final do path, você continuaria vulnerável a erros de digitação como `sl` ou `ks`.

- É inconsistente e pode ser confuso. Um ponto em um path pode significar que o comportamento de um comando irá mudar de acordo com o diretório corrente.

13.3.2 O path das páginas de manual

O path tradicional das páginas de manual era determinado pela variável de ambiente `MANPATH`, mas você não deve defini-la, pois fazer isso sobrescreverá os defaults do sistema em `/etc/manpath.config`.

13.3.3 O prompt

Os usuários experientes tendem a evitar prompts longos, complicados e inúteis. Em comparação, muitos administradores e algumas distribuições arrastam tudo para um prompt default. Sua opção deve refletir as necessidades de seus usuários; coloque o diretório de trabalho corrente, o nome do host e o nome do usuário no prompt se isso realmente ajudar.

Acima de tudo, evite caracteres que possam ser significativos para o shell, por exemplo:

```
{ } = & < >
```

 **Observação:** tome cuidado em especial para evitar o caractere `>`, que pode fazer com que arquivos vazios e erráticos apareçam em seu diretório corrente caso você acidentalmente copie e cole uma seção da janela de seu shell (lembre-se de que `>` redireciona a saída para um arquivo).

Mesmo um prompt default do shell pode não ser ideal. Por exemplo, o prompt default do `bash` contém o nome do shell e o número da versão.

Esse prompt simples definido para o `bash` termina com o costumeiro `$` (o prompt tradicional do `cs` termina com `%`):

```
PS1='\u\$ '
```

`\u` é um substituto para o nome do usuário corrente [veja a seção `PROMPTING` da página de manual `bash(1)`]. Outros substitutos populares incluem:

- `\h` – o nome do host (o formato compacto, sem nomes de domínio).
- `\!` – o número no histórico.
- `\w` – o diretório corrente. Como esse nome pode ser longo, é possível limitar a

exibição somente ao componente final usando \w.

- \\$ – \$ se a execução estiver ocorrendo com uma conta de usuário, # se for root.

13.3.4 Aliases

Entre os pontos mais complicados dos ambientes de usuário modernos está o papel dos *aliases* – um recurso do shell que provoca a substituição de uma string por outra antes da execução de um comando. Os aliases podem representar atalhos eficazes para economizar um pouco de digitação. Contudo eles também têm suas desvantagens:

- A manipulação de argumentos pode se tornar complicada.
- São confusos; um comando `which` incluído no shell pode informar se algo é um alias, porém não informará em que local ele está definido.
- São desconsiderados em subshells e em shells não interativos; não funcionam em outros shells.

Considerando essas desvantagens, provavelmente você deve evitar os aliases sempre que for possível, pois é mais fácil escrever uma função de shell ou um shell script totalmente novo. Os computadores modernos podem iniciar e executar shells tão rapidamente que a diferença entre um alias e um comando totalmente novo não deve significar nada para você.

Apesar disso, os aliases serão práticos se você quiser alterar uma parte do ambiente do shell. Não é possível alterar uma variável de ambiente com um shell script porque os scripts são executados como subshells. (Você também pode definir funções de shell para realizar essa tarefa.)

13.3.5 A máscara de permissões

Conforme descrito no capítulo 2, uma funcionalidade pronta do shell – o `umask` (máscara de permissões) – define suas permissões default. Você deve executar `umask` em um de seus arquivos de inicialização para garantir que qualquer programa que seja executado crie arquivos com as permissões que você deseja. As duas opções razoáveis são:

- 077 – essa é a máscara de permissões mais restritiva, pois não concede acesso a novos arquivos e diretórios a nenhum outro usuário. Normalmente, ela é apropriada em um sistema multiusuário, em que você não quer que outros usuários vejam nenhum de seus arquivos. No entanto, quando definida como default, essa máscara, às vezes, pode resultar em problemas quando seus usuários quiserem compartilhar arquivos, mas não souberem como definir as permissões corretamente. (Usuários inexperientes têm a tendência de configurar arquivos em modo de escrita global.)

- 022 – essa máscara concede acesso de leitura para novos arquivos e diretórios a outros usuários. Isso pode ser importante em um sistema monousuário, pois muitos daemons que executam como pseudousuários não serão capazes de ver arquivos e diretórios criados com o umask 077 mais restritivo.

👍 **Observação:** determinadas aplicações (em especial, programas de email) sobrescrevem o umask, alterando-o para 077, pois acham que seus arquivos não são de interesse de ninguém, exceto do dono do arquivo.

13.4 Ordem dos arquivos de inicialização e exemplos

Agora que você já sabe o que deve ser colocado nos arquivos de inicialização de shell, é hora de ver alguns exemplos específicos. De maneira surpreendente, uma das partes mais difíceis e confusas da criação de arquivos de inicialização é determinar qual dos vários arquivos deverá ser usado. As próximas seções discutem os dois shells Unix mais populares: o bash e o tcsh.

13.4.1 O shell bash

No bash, você pode escolher entre os arquivos de inicialização *.bash_profile*, *.profile*, *.bash_login* e *.bashrc*. Qual deles é apropriado para o seu path de comandos, o path das páginas de manual, o prompt, os aliases e a máscara de permissões? A resposta é que você deve ter um arquivo *.bashrc* acompanhado de um link simbólico *.bash_profile* que aponte para *.bashrc* porque há alguns tipos diferentes de instâncias de shell bash.

Os dois tipos principais de instância de shell são interativo e não interativo, porém, entre eles, somente os shells interativos interessam, pois os não interativos (como aqueles que executam shell scripts) geralmente não leem nenhum arquivo de inicialização. Os shells interativos são aqueles usados para executar comandos a partir de um terminal, por exemplo, aqueles que vimos neste livro, e eles podem ser classificados como shell de *login* ou de *não login*.

Shells de login

Tradicionalmente, um shell de login é aquele que você obtém quando faz login pela primeira vez em um sistema com o terminal usando um programa como */bin/login*. Fazer login remotamente com o SSH também faz com que você obtenha um shell de login. A ideia básica é que o shell de login é um shell inicial. Você pode dizer se um shell é de login ao executar `echo $0`; se o primeiro caractere for um -, o shell será de login.

Quando o bash executar como shell de login, ele executará */etc/profile*. Então ele

procurará os arquivos *.bash_profile*, *.bash_login* e *.profile* de um usuário, executando somente o primeiro que encontrar.

Por mais estranho que possa parecer, é possível executar um shell não interativo como um shell de login para forçá-lo a executar os arquivos de inicialização. Para isso, inicie o shell usando a opção `-l` ou `--login`.

Shells de não login

Um shell de não login é um shell adicional executado depois que você fizer login. É simplesmente qualquer shell interativo que não seja um shell de login. Programas de terminal relativos a sistemas de janelas (`xterm`, GNOME Terminal e assim por diante) iniciam shells de não login, a menos que você solicite especificamente um shell de login.

Na inicialização de um shell de não login, o `bash` executa */etc/bash.bashrc* e, em seguida, executa o *.bashrc* do usuário.

As consequências dos dois tipos de shell

O raciocínio por trás de dois sistemas de arquivos diferentes de inicialização é que, nos velhos tempos, os usuários faziam login por meio de um terminal tradicional com um shell de login e, em seguida, iniciavam subshells de não login com sistemas de janelas ou com o programa `screen`. Para os subshells de não login, configurar repetidamente o ambiente do usuário e executar vários programas que já haviam sido executados era considerado perda de tempo. Com os shells de login, era possível executar comandos sofisticados de inicialização em um arquivo como *.bash_profile*, deixando somente aliases e outros itens “leves” ao seu *.bashrc*.

Hoje em dia, a maioria dos usuários de desktop faz login por meio de um gerenciador de display gráfico (você aprenderá mais sobre eles no próximo capítulo). A maioria deles começa com um shell de login não interativo para preservar o modelo de login *versus* não login descrito anteriormente. Quando não o fizerem, será preciso configurar todo o seu ambiente (`path`, `path` do manual e assim por diante) em seu *.bashrc*, ou você não verá nenhum de seus ambientes nos shells de suas janelas de terminal. No entanto, o *.bash_profile* também será necessário se você quiser fazer login no console ou remotamente, pois esses shells de login não dão atenção ao *.bashrc*.

Exemplo de *.bashrc*

Para satisfazer tanto os shells de login quanto os shells de não login, como você criaria um *.bashrc* que também possa ser usado como o seu *.bash_profile*? Eis um exemplo

bem elementar (embora seja perfeitamente suficiente):

```
# Path de comandos
PATH=/usr/local/bin:/usr/bin:/bin:/usr/games
PATH=$HOME/bin:$PATH

# PS1 é o prompt normal
# As substituições incluem:
# \u nome do usuário \h nome do host \w diretório corrente
# \! número do histórico \s nome do shell \$ $ se usuário normal
PS1='\u\$ '

# EDITOR e VISUAL determinam o editor que programas como less
# e clientes de email chamam quando solicitados a editar um arquivo.
EDITOR=vi
VISUAL=vi

# PAGER é o visualizador default de arquivos-texto para programas como man.
PAGER=less

# Estas são algumas opções práticas para less.
# Um estilo diferente é LESS=FRX
# (F=sai no final, R=mostra caracteres puros, X=não usa alt screen)
LESS=meiX

# Você deve exportar variáveis de ambiente.
export PATH EDITOR VISUAL PAGER LESS

# Por padrão, concede acesso somente de leitura aos outros usuários para a maioria dos arquivos novos.
umask 022
```

Conforme descrito anteriormente, você pode compartilhar esse arquivo *.bashrc* com *.bash_profile* por meio de um link simbólico, ou pode tornar o relacionamento mais claro ainda ao criar *.bash_profile* com a linha única a seguir:

```
. $HOME/.bashrc
```

Verificando se os shells são de login e se são interativos

Com um *.bashrc* que coincida com seu *.bash_profile*, normalmente, você não executará comandos extras para os shells de login. No entanto, se você quiser definir ações diferentes para shells de login e de não login, o teste a seguir poderá ser adicionado ao seu *.bashrc*, que verifica a variável `$-` do shell em busca de um caractere `i`:

```
case $- in
  i*) # comandos interativos devem ser inseridos aqui
    comando
    --trecho omitido--
    ;;
  *) # comandos não interativos devem ser inseridos aqui
```



```
comando
--trecho omitido--
;;
esac
```

13.4.2 O shell tcsh

O `csh` padrão em praticamente todos os sistemas Linux é o `tcsh` – um shell C melhorado que popularizou funcionalidades como a edição de linhas de comando, nomes de arquivo com vários modos e preenchimento automático de comandos. Mesmo que o `tcsh` não seja usado como shell default para novos usuários (sugerimos usar o `bash`), você ainda deverá disponibilizar arquivos de inicialização para ele caso seus usuários venham a utilizá-lo.

Não é preciso se preocupar com a diferença entre shells de login e de não login no `tcsh`. Na inicialização, o `tcsh` procura um arquivo `.tcshrc`. Se esse arquivo não for encontrado, ele procurará o arquivo de inicialização `.cshrc` do shell `csh`. O motivo dessa ordem é que você pode usar o arquivo `.tcshrc` para extensões de `tcsh` que não funcionem com o `csh`. Provavelmente, você deverá se ater ao uso do `.cshrc` tradicional no lugar de `.tcshrc`; é bem pouco provável que alguém vá algum dia usar seus arquivos de inicialização com o `csh`. E se um usuário realmente se deparar com o `csh` em outro sistema, seu `.cshrc` funcionará.

Exemplo de .cshrc

Eis um exemplo de arquivo `.cshrc`:

```
# Path de comandos.
setenv PATH /usr/local/bin:/usr/bin:/bin:$HOME/bin

# EDITOR e VISUAL determinam o editor que programas como less
# e clientes de email chamam quando solicitados a editar um arquivo.
setenv EDITOR vi
setenv VISUAL vi

# PAGER é o visualizador default de arquivos-texto para programas como man.
setenv PAGER less

# Estas são algumas opções práticas para less.
setenv LESS meïX

# Por padrão, concede acesso somente de leitura aos outros usuários para a maioria dos arquivos novos.
umask 022

# Personaliza o prompt.
# As substituições incluem:
# %n nome do usuário %m nome do host %/ diretório corrente
```

```
# %h número no histórico %l terminal corrente %% %  
set prompt="%m%% " "
```

13.5 Configurações default do usuário

A melhor maneira de criar arquivos de inicialização e selecionar os defaults para os novos usuários é fazer experimentos com um novo usuário de teste no sistema. Crie o usuário de teste com um diretório home vazio e evite copiar seus próprios arquivos de configuração para o diretório do usuário de teste. Crie os novos arquivos de inicialização do zero.

Quando você achar que tem uma inicialização que funcione, faça login como o novo usuário de teste de todas as maneiras possíveis (no console, remotamente e assim por diante). Certifique-se de fazer o máximo possível de testes, incluindo testar a operação do sistema de janelas e as páginas de manual. Quando estiver satisfeito com o usuário de teste, crie um segundo usuário de teste copiando os arquivos de inicialização do primeiro usuário. Se tudo continuar funcionando, você agora terá um novo conjunto de arquivos de configuração que poderá ser distribuído aos novos usuários.

As seções a seguir apresentam defaults razoáveis para novos usuários.

13.5.1 Shells default

O shell default para qualquer novo usuário em um sistema Linux deve ser o `bash` porque:

- Os usuários interagem com o mesmo shell que usam para criar shell scripts (por exemplo, o `csh` é uma ferramenta notoriamente ruim para scripting – nem pense em usá-lo).
- O `bash` é padrão em sistemas Linux.
- O `bash` usa o GNU `readline` e, desse modo, sua interface é idêntica àquela de várias outras ferramentas.
- O `bash` oferece um controle adequado e fácil de compreender sobre redirecionamento de I/O e handles de arquivo.

No entanto muitos magos experientes do Unix usam shells como `csh` e `tesh` simplesmente porque não suportam mudar. É claro que você pode escolher o shell que quiser, porém escolha o `bash` se você não tiver nenhuma preferência e use-o como shell default para qualquer novo usuário do sistema. (Um usuário pode mudar seu shell usando o comando `chsh` para se adequar às preferências individuais.)

👉 **Observação:** há vários outros shells por aí (`rc`, `ksh`, `zsh`, `es` e assim por diante). Alguns não são

apropriados como shells para iniciantes, porém o `zsh` e o `fish` às vezes são populares entre os novos usuários que estiverem procurando um shell alternativo.

13.5.2 Editor

Em um sistema tradicional, o editor default deve ser o `vi` ou o `emacs`. Esses são os únicos editores que podemos praticamente garantir que estarão presentes em quase todo sistema Unix, o que significa que eles causarão o mínimo de problemas no longo prazo para um novo usuário. Entretanto as distribuições Linux, com frequência, configuram o `nano` como editor default, pois é mais fácil para os iniciantes usarem.

Como ocorre com os arquivos de inicialização de shell, evite arquivos grandes de inicialização de editores default. Um pequeno `set showmatch` no arquivo de inicialização `.exrc` não faz mal a ninguém, mas fique longe de qualquer configuração que altere significativamente o comportamento ou a aparência do editor, por exemplo, o recurso `showmode`, a indentação automática e as margens externas.

13.5.3 Pager

É perfeitamente razoável definir a variável de ambiente `PAGER` default com `less`.

13.6 Armadilhas dos arquivos de inicialização

Evite as ações a seguir em arquivos de inicialização:

- Não coloque nenhum tipo de comando gráfico em um arquivo de inicialização de shell.
- Não defina a variável de ambiente `DISPLAY` em um arquivo de inicialização de shell.
- Não defina o tipo de terminal em um arquivo de inicialização de shell.
- Não seja excessivamente econômico em comentários descritivos nos arquivos de inicialização default.
- Não execute comandos que exibam informações na saída-padrão em um arquivo de inicialização.
- Jamais defina `LD_LIBRARY_PATH` em um arquivo de inicialização de shell (veja a seção 15.1.4).

13.7 Tópicos adicionais relativos à inicialização

Como este livro trata somente do sistema Linux subjacente, não discutiremos os arquivos de inicialização do ambiente de janelas. Esse é um assunto bem amplo, pois o

gerenciador de display que permite fazer login em um sistema Linux moderno tem seu próprio conjunto de arquivos de inicialização, como *.xsession*, *.xinitrc*, além das combinações intermináveis de itens relacionados a GNOME e KDE.

As opções para sistemas de janelas podem parecer impressionantes, e não há nenhuma maneira comum de iniciar um ambiente de janelas no Linux. O próximo capítulo descreve algumas das várias possibilidades. Contudo, ao determinar o que seu sistema faz, você poderá se empolgar um pouco com os arquivos relacionados ao seu ambiente gráfico. Não há problemas nisso, porém não estenda essas configurações aos novos usuários. O mesmo princípio de manter a simplicidade nos arquivos de inicialização de shell funciona muito bem também para os arquivos de inicialização de GUI. Na verdade, é provável que você não vá precisar alterar nem um pouco os seus arquivos de inicialização de GUI.

CAPÍTULO 14

Uma breve análise do desktop Linux

Este capítulo contém uma apresentação breve dos componentes encontrados em um típico sistema desktop Linux. De todos os tipos diferentes de software que podem ser encontrados em sistemas Linux, o campo do desktop é um dos mais selvagens e variados, pois há muitos ambientes e aplicações entre os quais escolher, e a maioria das distribuições faz com que seja muito fácil experimentar usá-los.

De modo diferente de outras partes de um sistema Linux, como armazenamento e rede, não há muita hierarquia de camadas envolvidas na criação de uma estrutura de desktop. Em vez disso, cada componente realiza uma tarefa específica, comunicando-se com outros componentes à medida que for necessário. Alguns componentes compartilham blocos comuns de construção (em particular, as bibliotecas para kits de ferramenta gráficos), e esses blocos podem ser pensados como camadas simples de abstração, mas esse é o nível máximo de profundidade a que podemos chegar.

Este capítulo oferece uma discussão geral sobre os componentes de desktop, porém daremos uma olhada em dois elementos com um pouco mais de detalhes: o X Window System, que é a infraestrutura central por trás da maioria dos desktops, e o D-Bus, que é um serviço de comunicação entre processos usado em várias partes do sistema. Limitaremos a discussão e os exemplos práticos a alguns utilitários de diagnóstico que, embora não sejam incrivelmente úteis no dia a dia (a maioria das GUIs não exige que você dê comandos de shell para poder interagir com elas), ajudarão a entender o funcionamento subjacente do sistema e, quem sabe, proporcionarão um pouco de entretenimento ao longo do caminho. Também daremos uma olhada rapidamente na impressão.

14.1 Componentes do desktop

As configurações de desktop do Linux oferecem uma boa dose de flexibilidade. A maior parte do que é vivenciado pelos usuários de Linux (o “look and feel” do desktop) é proveniente de aplicações ou de blocos de construção de aplicações. Se você não gostar de uma aplicação em particular, normalmente será possível encontrar uma alternativa. E se o que você estiver procurando não existir, será possível escrevê-lo por

conta própria. Os desenvolvedores Linux tendem a ter uma ampla variedade de preferências em relação ao modo como o desktop deve atuar, o que resulta em muitas opções.

Para trabalhar em conjunto, todas as aplicações devem ter algo em comum, e, no centro de quase tudo na maioria dos desktops Linux, está o servidor X (X Window System). Pense no X como uma espécie de “kernel” do desktop, que administra tudo, desde renderizar janelas até configurar displays ou tratar dados de entrada de dispositivos como teclados e mouses. O servidor X também é um componente para o qual você não encontrará facilmente um substituto (veja a seção 14.4).

O servidor X é somente um servidor, e não determina o modo como nenhuma aplicação deve atuar ou se parecer. Em vez disso, os programas *clientes* do X cuidam da interface de usuário. Aplicações clientes básicas do X, como janelas de terminal e navegadores web, se conectam ao servidor X e pedem para que as janelas sejam desenhadas. Em resposta, o servidor X descobre onde colocar as janelas e as renderiza. O servidor X também envia os dados de entrada de volta ao cliente quando for apropriado.

14.1.1 Gerenciadores de janelas

Os clientes do X não precisam atuar como aplicações de usuário com janelas; eles podem atuar como serviços para outros clientes ou podem oferecer outras funções de interface. Um *gerenciador de janelas* talvez seja a aplicação de serviço cliente mais importante, pois ele descobre como organizar as janelas na tela e disponibiliza decorações interativas, como barras de título, que permitem ao usuário mover e minimizar as janelas. Esses elementos são fundamentais para a experiência do usuário.

Há várias implementações de gerenciadores de janelas. Exemplos como o Mutter/GNOME Shell e o Compiz foram criados para serem mais ou menos autônomos, enquanto outros estão incluídos em ambientes como o Xfce. A maioria dos gerenciadores de janelas incluída nas distribuições Linux padrão se esforça para oferecer conforto aos usuários, porém outros proporcionam efeitos visuais específicos ou adotam uma abordagem minimalista. Não é provável que algum dia vá haver um gerenciador de janelas padrão no Linux, pois as preferências e os requisitos dos usuários são diversificados e mudam constantemente; como resultado, novos gerenciadores de janelas surgem o tempo todo.

14.1.2 Kits de ferramentas

As aplicações desktop incluem determinados elementos comuns, como botões e menus, chamados *widgets*. Para agilizar o desenvolvimento e proporcionar uma aparência

comum, os programadores usam *kits de ferramentas* gráficas para disponibilizar esses elementos. Em sistemas operacionais como o Windows ou o Mac OS X, o fornecedor provê um kit de ferramentas comum, e a maioria dos programadores o utiliza. No Linux, o kit de ferramentas GTK+ é um dos mais comuns, porém, com frequência, você verá também widgets criados no framework Qt e em outros.

Os kits de ferramenta normalmente são constituídos de bibliotecas compartilhadas e arquivos de suporte, por exemplo, com imagens e informações de temas.

14.1.3 Ambientes desktop

Embora os kits de ferramenta proporcionem uma aparência externa uniforme aos usuários, alguns detalhes de um desktop exigem certo grau de cooperação entre as diferentes aplicações. Por exemplo, uma aplicação pode querer compartilhar dados com outra ou atualizar uma barra de notificação comum em um desktop. Para atender a essas necessidades, os kits de ferramentas e outras bibliotecas são agrupados em pacotes maiores chamados de *ambientes desktop*. GNOME, KDE, Unity e Xfce são alguns dos ambientes comuns de desktop Linux.

Os kits de ferramentas estão no centro da maioria dos ambientes desktop; porém, para criar um desktop uniforme, os ambientes também devem incluir vários arquivos de suporte, como ícones e configurações, que compõem os temas. Tudo isso é consolidado em documentos que descrevem as convenções de design, por exemplo, o modo como os menus e os títulos das aplicações devem ser mostrados e como as aplicações devem reagir a determinados eventos do sistema.

14.1.4 Aplicações

Acima do desktop estão as aplicações, por exemplo, os navegadores web e a janela do terminal. As aplicações X podem variar de básicas (como o antigo programa `xclock`) a complexas (como o navegador web Chrome e o pacote LibreOffice). Essas aplicações normalmente são independentes, mas, em geral, usam comunicação entre processos para tomarem conhecimento dos eventos pertinentes. Por exemplo, uma aplicação pode expressar interesse quando um novo dispositivo de armazenamento for conectado ou quando um novo email ou uma mensagem instantânea for recebida. Essa comunicação geralmente ocorre por meio do D-Bus, descrito na seção 14.5.

14.2 Um olhar mais detalhado sobre o X Window System

Historicamente, o X Window System (<http://www.x.org/>) é bem grande, com a

distribuição base incluindo o servidor X, bibliotecas de suporte a clientes e clientes. Em virtude do surgimento dos ambientes desktop como GNOME e KDE, a função da distribuição X mudou ao longo do tempo, com o foco atualmente mais no servidor principal que administra a renderização e os dispositivos de entrada assim como em uma biblioteca de cliente simplificada.

O servidor X é fácil de ser identificado em seu sistema. Ele se chama x. Verifique-o em uma listagem de processos; geralmente, você o verá sendo executado com várias opções como estas:

```
/usr/bin/X :0 -auth /var/run/lightdm/root/:0 -nolisten tcp vt7 -novtswitch
```

O :0 mostrado nesse caso chama-se *display* – um identificador que representa um ou mais monitores acessados com um teclado e/ou um mouse comuns. Geralmente, o display corresponde simplesmente ao monitor único conectado ao seu computador, porém é possível colocar vários monitores no mesmo display. Ao usar uma sessão X, a variável de ambiente DISPLAY é configurada com o identificador do display.

👍 **Observação:** os displays podem ser adicionalmente subdivididos em telas como :0.0 e :0.1, porém isso está se tornando cada vez mais raro porque as extensões do X, como o RandR, podem combinar vários monitores em uma tela virtual maior.

No Linux, um servidor X executa em um terminal virtual. Nesse exemplo, o argumento vt7 nos informa que o servidor foi especificado para executar em */dev/tty7* (normalmente, o servidor começa no primeiro terminal virtual disponível). Você pode executar mais de um servidor X ao mesmo tempo no Linux ao executá-los em terminais virtuais separados, porém, se você o fizer, cada servidor precisará de um identificador de display único. Podemos alternar entre os servidores usando as teclas Ctrl-Alt-Fn ou o comando chvt.

14.2.1 Gerenciadores de display

Normalmente, não iniciamos o servidor X com uma linha de comando porque iniciar o servidor não define nenhum cliente que deva executar no servidor. Se o servidor for iniciado por si só, você simplesmente obterá uma tela em branco. Em vez disso, a maneira mais comum de iniciar um servidor X é por meio de um *gerenciador de display* – um programa que inicia o servidor e apresenta uma caixa de login na tela. Ao fazer login, o gerenciador de display inicia um conjunto de clientes, por exemplo, um gerenciador de janelas e um gerenciador de arquivos, para que você possa começar a usar o computador.

Há vários gerenciadores de display diferentes como o gdm (para GNOME) e o kdm (para KDE). O lightdm na lista de argumentos da chamada anterior ao servidor X é um

gerenciador de display multiplataforma criado para ser capaz de iniciar sessões GNOME ou KDE.

Para iniciar uma sessão do X a partir de um console virtual em vez de usar um gerenciador de display, o comando `startx` ou `xinit` poderá ser executado. Entretanto a sessão que você obtiver provavelmente será bem simples e não se parecerá em nada com aquela obtida com um gerenciador de display, pois seu funcionamento e os arquivos de inicialização serão diferentes.

14.2.2 Transparência de rede

Um dos recursos do X é a transparência de rede. Como os clientes conversam com o servidor usando um protocolo, é possível executar clientes que se conectem a um servidor que esteja executando em um computador diferente diretamente pela rede; o servidor X fica ouvindo a porta 6000 à espera de conexões TCP. Os clientes que se conectarem a essa porta poderão se autenticar e, em seguida, poderão enviar janelas ao servidor.

Infelizmente, esse método, em geral, não oferece nenhuma criptografia e, como resultado, não é seguro. Para fechar essa brecha, a maioria das distribuições atualmente desabilita o listener de rede do servidor X (usando a opção `-nolisten tcp` do servidor). No entanto, ainda podemos executar clientes X de um computador remoto usando tunelamento SSH, conforme descrito no capítulo 10, ao conectar o socket de domínio Unix do servidor X a um socket no computador remoto.

14.3 Explorando os clientes do X

Embora uma pessoa normalmente não pense em trabalhar com uma interface gráfica de usuário a partir da linha de comando, há diversos utilitários que permitem explorar as partes do X Window System. Em particular, podemos inspecionar os clientes à medida que eles executarem.

Uma das ferramentas mais simples é o `xwininfo`. Quando executado sem argumentos, ele pedirá que você clique em uma janela:

```
$ xwininfo
```

```
xwininfo: Please select the window about which you
would like information by clicking the
mouse in that window.
```

Depois que você clicar, uma lista de informações sobre a janela será exibida, por exemplo, sua localização e o tamanho:

```
xwininfo: Window id: 0x5400024 "xterm"
```

```
Absolute upper-left X: 1075
```

```
Absolute upper-left Y: 594
```

```
--trecho omitido--
```

Observe o ID da janela nesse caso – o servidor X e os gerenciadores de janela usam esse identificador para monitorar as janelas. Para obter uma lista de todos os IDs de janela e dos clientes, utilize o comando `xlsclients -l`.

👍 **Observação:** há uma janela especial chamada janela raiz (root window), que é o plano de fundo (background) do display. No entanto, pode ser que você jamais veja essa janela (consulte a seção “Plano de fundo do desktop”).

14.3.1 Eventos do X

Os clientes do X obtêm seus dados de entrada e outras informações sobre o estado do servidor por meio de um sistema de eventos. Os eventos do X funcionam como outros eventos assíncronos de comunicação entre processos, como os eventos udev e os eventos D-Bus: o servidor X recebe informações de uma origem, por exemplo, de um dispositivo de entrada e, em seguida, redistribui essa entrada na forma de um evento para qualquer cliente do X que estiver interessado.

Você pode fazer experiências com eventos usando o comando `xev`. Executá-lo fará com que uma nova janela seja aberta, na qual você poderá mover o mouse, clicar e digitar. À medida que fizer isso, o `xev` irá gerar dados de saída descrevendo os eventos do X recebidos do servidor. A seguir, apresentamos um exemplo de saída para o movimento do mouse:

```
$ xev
```

```
--trecho omitido--
```

```
MotionNotify event, serial 36, synthetic NO, window 0x6800001,  
  root 0xbb, subw 0x0, time 43937883, (47,174), root:(1692,486),  
  state 0x0, is_hint 0, same_screen YES
```

```
MotionNotify event, serial 36, synthetic NO, window 0x6800001,  
  root 0xbb, subw 0x0, time 43937891, (43,177), root:(1688,489),  
  state 0x0, is_hint 0, same_screen YES
```

Observe as coordenadas entre parênteses. O primeiro par representa as coordenadas x e y do ponteiro do mouse na janela, e o segundo (root:) é a localização do ponteiro em todo o display.

Outros eventos de baixo nível incluem pressionamento de teclas e cliques de botão, porém alguns eventos mais sofisticados indicam se o mouse entrou ou saiu da janela, ou se a janela ganhou ou perdeu foco do gerenciador de janelas. Por exemplo, a seguir,

estão os eventos correspondentes à saída e perda de foco:

```
LeaveNotify event, serial 36, synthetic NO, window 0x6800001,  
  root 0xbb, subw 0x0, time 44348653, (55,185), root:(1679,420),  
  mode NotifyNormal, detail NotifyNonlinear, same_screen YES,  
  focus YES, state 0
```

```
FocusOut event, serial 36, synthetic NO, window 0x6800001,  
  mode NotifyNormal, detail NotifyNonlinear
```

Um uso comum de `xev` está em extrair códigos e símbolos de teclas para diferentes teclados ao remapeá-los. A seguir, apresentamos a saída referente ao pressionamento da tecla `L`; o código da tecla, nesse caso, é 46:

```
KeyPress event, serial 32, synthetic NO, window 0x4c00001,  
  root 0xbb, subw 0x0, time 2084270084, (131,120), root:(197,172),  
  state 0x0, keycode 46 (keysym 0x6c, l), same_screen YES,  
  XLookupString gives 1 bytes: (6c) "l"  
  XmbLookupString gives 1 bytes: (6c) "l"  
  XFilterEvent returns: False
```

O `xev` também pode ser associado a um ID de janela existente por meio da opção `-id id`. (Use o ID obtido de `xwininfo` como `id` ou monitore a janela raiz com `-root`.)

14.3.2 Entendendo as entradas do X e as configurações de preferências

Uma das características mais potencialmente impressionantes do X está no fato de normalmente haver mais de uma maneira de configurar as preferências, e alguns métodos podem não funcionar. Por exemplo, uma preferência comum de teclado em sistemas Linux consiste em remapear a tecla Caps Lock para uma tecla Control. Há várias maneiras de fazer isso, desde pequenos ajustes com o antigo comando `xmodmap` até fornecer um mapa totalmente novo de teclado usando o utilitário `setxkbmap`. Como você saberá qual dos métodos deverá usar (se é que um deles deverá ser usado)? É uma questão de saber quais partes do sistema têm a responsabilidade, porém determinar isso pode ser difícil. Tenha em mente que um ambiente desktop pode prover suas próprias configurações e sobreposições.

Com isso esclarecido, a seguir, apresentaremos algumas referências sobre a infraestrutura subjacente.

Dispositivos de entrada (geral)

O servidor X usa o *X Input Extension* para administrar os dados de entrada de vários dispositivos diferentes. Há dois tipos básicos de dispositivos de entrada – teclado e

ponteiro (mouse) –, e você pode associar quantos dispositivos quiser. Para usar mais de um dispositivo do mesmo tipo simultaneamente, o X Input Extension cria um dispositivo “virtual core” que canaliza os dados de entrada do dispositivo para o servidor X. O dispositivo core se chama mestre (master); os dispositivos físicos conectados ao computador se tornam escravos (slaves).

Para ver a configuração dos dispositivos em seu computador, experimente executar o comando `xinput --list`:

```
$ xinput --list
```

```
Virtual core pointer          id=2  [master pointer (3)]
  Virtual core XTEST pointer  id=4  [slave pointer (2)]
  Logitech Unifying Device    id=8  [slave pointer (2)]
Virtual core keyboard         id=3  [master keyboard (2)]
  Virtual core XTEST keyboard id=5  [slave keyboard (3)]
  Power Button                id=6  [slave keyboard (3)]
  Power Button                id=7  [slave keyboard (3)]
  Cypress USB Keyboard        id=9  [slave keyboard (3)]
```

Cada dispositivo tem um ID associado que pode ser usado com `xinput` e com outros comandos. Nessa saída, os IDs 2 e 3 são os dispositivos core, e os IDs 8 e 9 são os dispositivos reais. Observe que os botões de alimentação (power buttons) do computador também são tratados como dispositivos de entrada do X.

A maioria dos clientes do X espera dados de entrada dos dispositivos core, pois não há nenhum motivo para eles se preocuparem com o dispositivo em particular que originou um evento. Com efeito, a maioria dos clientes não sabe nada sobre o X Input Extension. Contudo um cliente pode usar a extensão para selecionar um dispositivo em particular.

Cada dispositivo tem um conjunto associado de *propriedades*. Para visualizar as propriedades, utilize `xinput` com o número do dispositivo, como mostrado no exemplo a seguir:

```
$ xinput --list-props 8
```

```
Device 'Logitech Unifying Device. Wireless PID:4026':
  Device Enabled (126):  1
  Coordinate Transformation Matrix (128): 1.000000, 0.000000, 0.000000,
0.000000, 1.000000, 0.000000, 0.000000, 0.000000, 0.000000, 1.000000
  Device Accel Profile (256):  0
  Device Accel Constant Deceleration (257):  1.000000
  Device Accel Adaptive Deceleration (258):  1.000000
  Device Accel Velocity Scaling (259):  10.000000
```

```
--trecho omitido--
```

Como você pode ver, há várias propriedades muito interessantes que podem ser

alteradas com a opção `--set-prop`. (Consulte a página de manual `xinput` (1) para obter mais informações.)

Mouse

Podemos manipular configurações relacionadas a dispositivos usando o comando `xinput`, e muitas das configurações mais úteis dizem respeito ao mouse (ponteiro). Várias configurações podem ser alteradas diretamente como propriedades, porém, normalmente, é mais fácil usar as opções especializadas `--set-ptr-feedback` e `--set-button-map` do `xinput`. Por exemplo, se você tiver um mouse de três botões em *disp* e quiser inverter a ordem dos botões (isso é prático para usuários canhotos), experimente executar isto:

```
$ xinput --set-button-map disp 3 2 1
```

Teclado

Os vários layouts diferentes de teclado disponíveis internacionalmente resultam em dificuldades particulares para integração em qualquer sistema de janelas. O X sempre teve um recurso de mapeamento interno de teclado em seu protocolo central, que pode ser manipulado com o comando `xmodmap`, porém qualquer sistema razoavelmente moderno usa o XKB (a extensão de teclado do X) para ter um controle mais preciso.

O XKB é complicado, a ponto de muitas pessoas continuarem usando o `xmodmap` quando precisam fazer alterações rápidas. A ideia básica por trás do XKB é que você pode definir um mapa de teclado e compilá-lo com o comando `xkbcomp` e, em seguida, carregar e ativar esse mapa no servidor X com o comando `setxkbmap`. Dois recursos especialmente interessantes do sistema são:

- A capacidade de definir mapas parciais para complementar os mapas existentes. Isso é especialmente prático para tarefas como mudar sua tecla Caps Lock para uma tecla Control, e é usado por vários utilitários gráficos de preferência de teclado em ambientes desktop.
- A capacidade de definir mapas individuais para cada teclado conectado.

Plano de fundo do desktop

O antigo comando `xsetroot` do X permite definir a cor do plano de fundo e outras características da janela raiz, porém não terá efeito na maioria dos computadores porque a janela raiz nunca está visível. Em vez disso, a maioria dos ambientes desktop coloca uma janela grande atrás de todas as demais janelas para permitir funcionalidades como um “papel de parede ativo” e navegação em arquivos do desktop. Há maneiras de alterar o plano de fundo a partir da linha de comando (por exemplo,

com o comando `gsettings` em algumas instalações GNOME), porém, se você realmente *quiser* fazer isso, é provável que você esteja com muito tempo disponível.

xset

Provavelmente, o comando mais antigo de preferência é o `xset`. Ele não é muito mais usado atualmente, porém, um `xset q` pode ser executado rapidamente para obter o status de alguns recursos. Talvez os mais úteis sejam as configurações de protetor de tela (screensaver) e o *DPMS* (Display Power Management Signaling).

14.4 O futuro do X

À medida que você lia a discussão anterior, talvez você tenha tido a impressão de que o X é um sistema realmente antigo, que vem sendo bastante alterado para que consiga realizar novos truques. Você não estaria tão longe assim da verdade. O X Window System foi inicialmente desenvolvido nos anos 80. Embora sua evolução ao longo dos anos tenha sido significativa (a flexibilidade era uma parte importante de seu design inicial), a arquitetura original pode ser forçada somente até certo ponto.

Um sinal da idade do X Window System é que o próprio servidor suporta uma quantidade extremamente grande de bibliotecas, muitas para compatibilidade com versões anteriores. Talvez, porém, o aspecto mais significativo seja a ideia de que ter um servidor administrando os clientes e suas janelas e atuando como intermediário para a memória das janelas tenha se tornado um peso para o desempenho. É muito mais rápido permitir que as aplicações renderizem o conteúdo de suas janelas diretamente na memória do display, com um gerenciador de janelas mais leve chamado *compositing window manager* (gerenciador de janelas compostas) para organizar as janelas e realizar um mínimo de gerenciamento da memória do display.

Um novo padrão baseado nessa ideia – o Wayland – começou a ganhar força. A parte mais significativa do Wayland é um protocolo que define como os clientes conversam com o *compositing window manager*. Outras partes incluem o gerenciamento dos dispositivos de entrada e um sistema de compatibilidade com o X. Como protocolo, o Wayland também mantém a ideia de transparência de rede. Muitas partes do desktop Linux atualmente suportam o Wayland, como o GNOME e o KDE.

Porém o Wayland não é a única alternativa ao X. Na época desta publicação, outro projeto – o Mir – tinha objetivos semelhantes, embora sua arquitetura adotasse uma abordagem, de certo modo, diferente. Em algum momento, haverá uma ampla adoção de pelo menos um sistema, que poderá ou não ser um deles.

Esses novos desenvolvimentos são significativos porque não estão limitados ao desktop Linux. Em virtude do desempenho pobre e do enorme uso de memória, o X Window System não é adequado para ambientes como tablets e smartphones, de modo que os fabricantes, até agora, têm usado sistemas alternativos para tratar os displays Linux embarcados. Entretanto padronizar a renderização direta pode oferecer uma maneira mais eficiente de suportar esses displays no que diz respeito ao custo.

14.5 D-Bus

Um dos desenvolvimentos mais importantes a surgir do desktop Linux é o *Desktop Bus (D-Bus)* – um sistema para envio de mensagens. O D-Bus é importante porque serve como um sistema de comunicação entre processos que permite às aplicações desktop conversarem umas com as outras, e porque a maioria dos sistemas Linux o utiliza para notificar os processos a respeito dos eventos do sistema, por exemplo, a inserção de um drive USB.

O D-Bus propriamente dito é constituído de uma biblioteca que padroniza a comunicação entre processos com um protocolo e suporta funções para quaisquer dois processos conversarem um com o outro. Por si só, essa biblioteca não oferece muito mais que uma versão elegante dos recursos normais de IPC, por exemplo, os sockets de domínio Unix. O que torna o D-Bus útil é a existência de um “hub” central chamado `dbus-daemon`. Os processos que devem reagir a eventos podem se conectar ao `dbus-daemon` e se registrar para receber determinados tipos de evento. Os processos também criam eventos. Por exemplo, o processo `udisks-daemon` ouve o `ubus` em busca de eventos de disco e os envia ao `dbus-daemon`, que, por sua vez, retransmite os eventos às aplicações interessadas em eventos de disco.

14.5.1 Instâncias de sistema e de sessão

O D-Bus tornou-se uma parte mais integrante do sistema Linux e, atualmente, vai além do desktop. Por exemplo, tanto o `systemd` quanto o `Upstart` têm canais D-Bus de comunicação. Entretanto adicionar dependências de ferramentas de desktop no core (núcleo) do sistema vai contra uma regra de design do core do Linux.

Para resolver esse problema, há dois tipos de instâncias (processos) de `dbus-daemon` que podem ser executados. O primeiro é a instância de sistema, que é iniciada pelo `init` no momento do boot, com a opção `--system`. A instância de sistema normalmente é executada como um usuário D-Bus, e seu arquivo de configuração é `/etc/dbus-1/system.conf` (embora você provavelmente não deva alterar a configuração). Os processos podem se

conectar à instância do sistema por meio do socket de domínio Unix `/var/run/dbus/system_bus_socket`.

Independentemente da instância de sistema do D-Bus, há uma instância de sessão opcional que é executada somente quando uma sessão desktop é iniciada. As aplicações desktop que você executar se conectarão a essa instância.

14.5.2 Monitorando as mensagens do D-Bus

Uma das melhores maneiras de ver a diferença entre as instâncias de `dbus-daemon` de sistema e de sessão é monitorar os eventos que passam pelo bus. Experimente usar o utilitário `dbus-monitor` em modo sistema da seguinte maneira:

```
$ dbus-monitor --system
signal sender=org.freedesktop.DBus -> dest=:1.952 serial=2 path=/org/
freedesktop/DBus; interface=org.freedesktop.DBus; member=NameAcquired
string ":1.952"
```

A mensagem de inicialização, nesse caso, indica que o monitor está conectado e que adquiriu um nome. Você não deverá ver muita atividade ao executá-lo dessa maneira porque a instância de sistema normalmente não está muito ocupada. Para ver algo acontecer, experimente conectar um dispositivo de armazenamento USB.

Em comparação, as instâncias de sessão têm muito mais a fazer. Supondo que você tenha feito login em uma sessão de desktop, experimente executar:

```
$ dbus-monitor --session
```

Agora mova o seu mouse por diferentes janelas; se seu desktop tiver conhecimento do D-Bus, você deverá obter uma chuva de mensagens indicando janelas ativadas.

14.6 Impressão

Imprimir um documento no Linux é um processo de vários estágios. Esse processo funciona da seguinte maneira:

1. O programa efetuando a impressão normalmente converte o documento para o formato PostScript. Esse passo é opcional.
2. O programa envia o documento para um servidor de impressão.
3. O servidor de impressão recebe o documento e o coloca em uma fila de impressão.
4. Quando a vez do documento na fila chegar, o servidor de impressão enviará o documento para um filtro de impressão.
5. Se o documento não estiver em formato PostScript, um filtro de impressão poderá

realizar uma conversão.

6. Se a impressora destino não entender PostScript, um driver de impressora converterá o documento para um formato compatível com a impressora.
7. O driver de impressora adiciona instruções opcionais ao documento, por exemplo, opções de bandeja de papel e impressão dos dois lados.
8. O servidor de impressão usa um backend para enviar o documento para a impressora.

A parte mais confusa desse processo é o motivo pelo qual ele gira bastante em torno do PostScript. O PostScript, na verdade, é uma linguagem de programação, portanto, ao imprimir um arquivo usando-o, você envia um programa para a impressora. O PostScript serve como um padrão para impressão em sistemas do tipo Unix, de modo muito parecido como o formato *.tar* serve como um padrão de arquivamento. (Algumas aplicações atualmente usam uma saída PDF, porém ela é relativamente fácil de converter.)

Falaremos sobre o formato de impressão mais adiante; inicialmente, vamos dar uma olhada no sistema de filas.

14.6.1 CUPS

O sistema-padrão de impressão no Linux é o *CUPS* (<http://www.cups.org/>), que é o mesmo sistema usado no Mac OS X. O daemon do servidor CUPS chama-se *cupsd*, e o comando *lpr* pode ser usado como um cliente simples para enviar arquivos ao daemon.

Um recurso significativo do CUPS é que ele implementa o *IPP* (Internet Print Protocol, ou Protocolo de impressão para Internet) – um sistema que permite transações do tipo HTTP entre clientes e servidores na porta TCP 631. Com efeito, se o CUPS estiver executando em seu sistema, provavelmente você poderá se conectar a <http://localhost:631/> para ver sua configuração atual e verificar qualquer tarefa de impressão. A maioria das impressoras de rede e dos servidores de impressão suporta IPP, assim como o Windows, o que pode tornar relativamente simples a tarefa de configurar impressoras remotas.

É provável que você não vá poder administrar o sistema a partir da interface web, pois a configuração default não é muito segura. Em vez disso, sua distribuição provavelmente terá uma interface de configuração gráfica para adicionar e modificar impressoras. Essas ferramentas manipulam os arquivos de configuração, normalmente encontrados em */etc/cups*. Em geral, é melhor deixar essas ferramentas fazerem o trabalho para você, pois a configuração pode ser complicada. E mesmo que você se

depare com um problema e precise fazer a configuração manualmente, em geral, é melhor criar uma impressora usando as ferramentas gráficas para que você possa ter um ponto de partida.

14.6.2 Conversão de formatos e filtros de impressão

Muitas impressoras, incluindo quase todos os modelos mais simples, não entendem PostScript nem PDF. Para que o Linux suporte uma dessas impressoras, os documentos deverão ser convertidos para um formato específico da impressora. O CUPS envia o documento para um RIP (Raster Image Processor) para gerar um bitmap. O RIP quase sempre usa o programa Ghostscript (`gs`) para fazer a maior parte do verdadeiro trabalho, porém é um pouco complicado, pois o bitmap deve se adequar ao formato da impressora. Sendo assim, os drivers de impressora utilizados pelo CUPS consultam o arquivo PPD (PostScript Printer Definition) para a impressora específica a fim de determinar as configurações, por exemplo, a resolução e os tamanhos do papel.

14.7 Outros assuntos relacionados ao desktop

Uma característica interessante do ambiente desktop Linux está no fato de geralmente ser possível escolher quais partes você quer usar e parar de usar aquelas de que você não gosta. Para uma análise de vários projetos desktop, dê uma olhada nas listas de correspondência e nos links de projetos para os diversos projetos em <http://www.freedesktop.org/>. Em outros locais, você encontrará diferentes projetos para desktop como Ayatana, Unity e Mir.

Outro desenvolvimento importante no desktop Linux é o projeto de código aberto Chromium OS e sua contrapartida, o Google Chrome OS, encontrado nos PCs Chromebook. É um sistema Linux que usa boa parte da tecnologia de desktop descrita neste capítulo, porém está centrado em torno dos navegadores web Chromium/Chrome. Muito do que é encontrado em um desktop tradicional foi removido do Chrome OS.

CAPÍTULO 15

Ferramentas de desenvolvimento

O Linux e o Unix são bem populares entre os programadores não só por causa do incrível conjunto de ferramentas e de ambientes disponíveis, mas também porque o sistema é excepcionalmente bem documentado e transparente. Em um computador Linux, não é preciso ser um programador para tirar vantagem das ferramentas de desenvolvimento, mas, quando estiver trabalhando com o sistema, você deverá conhecer um pouco das ferramentas de programação, pois elas têm um papel mais importante no gerenciamento de sistemas Unix do que em outros sistemas operacionais. No mínimo, você deverá ser capaz de identificar os utilitários de desenvolvimento e ter alguma ideia de como executá-los.

Este capítulo reúne muitas informações em um espaço pequeno, mas não será preciso dominar tudo aqui. Você pode facilmente passar os olhos pelo material e retornar mais tarde. A discussão sobre bibliotecas compartilhadas provavelmente é a mais importante. Porém, para entender de onde vêm as bibliotecas compartilhadas, inicialmente será preciso ter algumas informações anteriores sobre como criar programas.

15.1 O compilador C

Saber como executar o compilador da linguagem de programação C pode dar uma boa ideia da origem dos programas que você vê em seu sistema Linux. O código-fonte da maioria dos utilitários Linux e de muitas aplicações em sistemas Linux está escrito em C ou em C++. Usaremos exemplos principalmente em C neste capítulo, porém as informações podem ser aplicadas ao C++.

Os programas C seguem um processo tradicional de desenvolvimento: os programas são escritos, compilados e executados. Isso quer dizer que, quando você escrever um programa C e quiser executá-lo, será necessário *compilar* o código-fonte escrito, gerando um formato binário de baixo nível que o computador entenda. Isso pode ser comparado às linguagens de scripting que serão discutidas mais adiante, em que não há necessidade de compilar nada.

👉 **Observação:** por padrão, a maioria das distribuições não inclui as ferramentas necessárias para compilar códigos C, pois essas ferramentas ocupam uma quantidade bem razoável de espaço. Se não puder encontrar algumas das ferramentas descritas aqui, você poderá instalar o pacote build-essential do Debian/Ubuntu ou executar o yum para o groupinstall “Development Tools” (Ferramentas de desenvolvimento) no Fedora/CentOS. Se isso não funcionar, tente procurar um pacote contendo “C compiler” (Compilador C).

O executável do compilador C na maioria dos sistemas Unix é o compilador GNU C – o gcc –, embora o compilador clang mais recente do projeto LLVM esteja ganhando popularidade. Os nomes dos arquivos de código-fonte C terminam com *.c*. Dê uma olhada no arquivo de código-fonte C único e autocontido chamado *hello.c*, que pode ser encontrado no livro *C: a linguagem de programação – padrão Ansi* de Brian W. Kernighan e Dennis M. Ritchie (Editora Campus, 1989):

```
#include <stdio.h>

main() {
    printf("Hello, World.\n");
}
```

Coloque esse código-fonte em um arquivo chamado *hello.c* e, em seguida, execute o comando:

```
$ cc hello.c
```

O resultado será um executável chamado *a.out*, que pode ser executado como qualquer outro executável do sistema. Entretanto você deveria dar outro nome ao executável (por exemplo, *hello*). Para isso, utilize a opção *-o* do compilador:

```
$ cc -o hello hello.c
```

Para programas pequenos, não há muito mais do que isso no que concerne à compilação. Talvez seja necessário adicionar um diretório extra de inclusão ou bibliotecas (veja as seções 15.1.2 e 15.1.3), mas vamos dar uma olhada em programas um pouco maiores antes de entrar nesses assuntos.

15.1.1 Vários arquivos-fonte

A maioria dos programas C são grandes demais para caberem de forma razoável em um único arquivo de código-fonte. Arquivos monstruosos se tornam muito desorganizados para o programador e, às vezes, os compiladores até mesmo têm problemas para interpretá-los. Desse modo, os desenvolvedores agrupam componentes do código-fonte, cada parte ficando com seu próprio arquivo.

Ao compilar a maioria dos arquivos *.c*, não crie um executável imediatamente. Em vez disso, utilize a opção *-c* do compilador em cada arquivo para criar *arquivos-objeto*. Para ver como isso funciona, vamos supor que você tenha dois arquivos: *main.c* e

aux.c. Os dois comandos de compilador a seguir fazem a maior parte do trabalho de gerar o programa:

```
$ cc -c main.c
```

```
$ cc -c aux.c
```

Os dois comandos anteriores compilam os dois arquivos de código-fonte, gerando dois arquivos-objeto: *main.o* e *aux.o*.

Um arquivo-objeto é um arquivo binário que um processador quase consegue entender, exceto pelo fato de ainda haver algumas pendências. Em primeiro lugar, o sistema operacional não sabe como executar um arquivo-objeto e, em segundo, é provável que você tenha de combinar vários arquivos-objeto e algumas bibliotecas de sistema para compor um programa completo.

Para gerar um executável totalmente funcional a partir de um ou mais arquivos-objeto, execute o *linker* – o comando *ld* no Unix. Os programadores raramente usam o *ld* na linha de comando, pois o compilador C sabe como executar o linker. Portanto, para criar um executável chamado *myprog* a partir dos dois arquivos-objeto anteriores, execute o comando a seguir para ligá-los:

```
$ cc -o myprog main.o aux.o
```

Embora seja possível compilar vários arquivos-fonte manualmente, como mostrado no exemplo anterior, poderá ser difícil manter o controle sobre todos eles durante o processo de compilação quando a quantidade de arquivos-fonte se multiplicar. O sistema *make*, descrito na seção 15.2, é o padrão Unix tradicional para gerenciamento de compilação. Esse sistema é especialmente importante no gerenciamento dos arquivos descritos nas duas próximas seções.

15.1.2 Arquivos de cabeçalho (include) e diretórios

Os *arquivos de cabeçalho* em C (header files) são arquivos adicionais de código-fonte que normalmente contêm declarações de tipo e de funções de biblioteca. Por exemplo, *stdio.h* é um arquivo de cabeçalho (veja o programa simples na seção 15.1).

Infelizmente, um grande número de problemas de compilador surge com os arquivos de cabeçalho. A maioria dos problemas ocorre quando o compilador não consegue achar esses arquivos e as bibliotecas. Há até mesmo alguns casos em que um programador se esquece de incluir um arquivo de cabeçalho necessário, fazendo com que parte do código-fonte não compile.

Corrigindo problemas de inclusão de arquivos

Localizar os arquivos de inclusão corretos nem sempre é fácil. Às vezes, há vários arquivos de inclusão com os mesmos nomes em diretórios diferentes, e não está claro qual deles é o arquivo correto. Quando o compilador não consegue encontrar um arquivo de inclusão, a mensagem de erro apresenta o seguinte aspecto:

```
badinclude.c:1:22: fatal error: notfound.h: No such file or directory
```

Essa mensagem informa que o compilador não pôde encontrar o arquivo de cabeçalho *notfound.h* que o arquivo *badinclude.c* referencia. Esse erro específico é resultado direto da diretiva a seguir, que está na linha 1 de *badinclude.c*:

```
#include <notfound.h>
```

O diretório de inclusão default no Unix é */usr/include*; o compilador sempre procura o arquivo de inclusão nesse local, a menos que você lhe diga explicitamente para não fazer isso. Entretanto, podemos fazer o compilador procurar em outros diretórios de inclusão (a maioria dos paths que contêm arquivos de cabeçalho têm *include* em algum ponto de seus nomes).

👍 **Observação:** você aprenderá mais sobre como encontrar arquivos de inclusão ausentes no capítulo 16.

Por exemplo, vamos supor que você tenha encontrado *notfound.h* em */usr/junk/include*. Podemos fazer o compilador ver esse diretório usando a opção *-I*:

```
$ cc -c -I/usr/junk/include badinclude.c
```

O compilador não deverá mais ter problemas com a linha de código em *badinclude.c* que faz referência ao arquivo de cabeçalho.

Tome cuidado também com inclusões que usem aspas duplas (" ") no lugar de sinais de menor e de maior (<>), desta maneira:

```
#include "myheader.h"
```

Aspas duplas significam que o arquivo de cabeçalho não está em um diretório de inclusão do sistema e que o compilador deve procurar no path de inclusão. Com frequência, significa que o arquivo de inclusão está no mesmo diretório que o arquivo-fonte. Se houver algum problema com aspas duplas, provavelmente você estará tentando compilar um código-fonte incompleto.

O que é o pré-processador C (cpp)?

O fato é que o compilador C não faz realmente o trabalho de procurar todos esses arquivos de inclusão. Essa tarefa cabe ao *pré-processador C* – um programa que o compilador executa em seu código-fonte antes de efetuar o parsing do programa. O pré-processador reescreve o código-fonte em um formato que o compilador entenda; é uma ferramenta para deixar o código-fonte mais fácil de ser lido (e para prover atalhos).

Os comandos do pré-processador no código-fonte são chamados de *diretivas*, e elas começam com o caractere #. Há três tipos básicos de diretivas:

- Arquivos de inclusão – uma diretiva `#include` instrui o pré-processador a incluir um arquivo inteiro. Observe que a flag `-I` do compilador, na realidade, é uma opção que faz o pré-processador procurar arquivos de inclusão em um diretório especificado, como vimos na seção anterior.
- Definições de macros – uma linha como `#define BLAH something` diz ao pré-processador para substituir todas as ocorrências de `BLAH` por `something` no código-fonte. A convenção determina que as macros tenham somente letras maiúsculas, porém você não deverá se surpreender com o fato de que, às vezes, os programadores usem macros cujos nomes se parecem com funções e variáveis. (Ocasionalmente, isso provoca muitas dores de cabeça. Vários programadores costumam abusar do pré-processador,)

👍 **Observação:** em vez de definir macros em seu código-fonte, elas também podem ser definidas por meio de passagem de parâmetros ao compilador: `-DBLAH=something` funciona como a diretiva anterior.

- Condicionais: determinadas partes do código podem ser marcadas com `#ifdef`, `#if` e `#endif`. A diretiva `#ifdef MACRO` verifica se a macro `MACRO` do pré-processador está definida, e `#if condição` testa se *condição* é diferente de zero. Em ambas as diretivas, se a condição que se seguir à “instrução if” for falsa, o pré-processador não passará nenhum texto de programa entre `#if` e o próximo `#endif` ao compilador. Se você planeja dar uma olhada em algum código C, é melhor se acostumar com isso.

Um exemplo de uma diretiva condicional será apresentada a seguir. Quando vir o código a seguir, o pré-processador verificará se a macro `DEBUG` está definida e, em caso afirmativo, passará a linha contendo `fprintf()` para o compilador. Caso contrário, o pré-processador pulará essa linha e continuará a processar o arquivo após o `#endif`:

```
#ifdef DEBUG
    fprintf(stderr, "This is a debugging message.\n");
#endif
```

👍 **Observação:** o pré-processador C não sabe nada sobre a sintaxe C, as variáveis, as funções e outros elementos. Ele entende somente suas próprias macros e diretivas.

No Unix, o nome do pré-processador C é `cpp`, porém ele também pode ser executado com `gcc -E`. No entanto, raramente será necessário executar o pré-processador sozinho.

15.1.3 Fazendo a ligação com bibliotecas

O compilador C não sabe o suficiente sobre o seu sistema para criar um programa útil

por si só. Você precisa de *bibliotecas* para criar programas completos. Uma biblioteca C é uma coleção de funções comuns pré-compiladas que podem ser incluídas em seu programa. Por exemplo, muitos executáveis usam a biblioteca de matemática, pois ela disponibiliza funções trigonométricas e recursos desse tipo.

As bibliotecas entram em cena principalmente no momento da ligação, quando o linker cria um executável a partir de arquivos-objeto. Por exemplo, se você tiver um programa que use a biblioteca gobject, mas se esquecer de dizer ao compilador para fazer a ligação com essa biblioteca, você verá erros de ligação como este:

```
badobject.o(.text+0x28): undefined reference to 'g_object_new'
```

As partes mais importantes dessa mensagem de erro estão em negrito. Quando analisou o arquivo-objeto *badobject.o*, o linker não pôde encontrar a função que aparece em negrito e, como consequência, não pôde criar o executável. Nesse caso em particular, você poderá suspeitar que se esqueceu da biblioteca gobject, pois a função que está faltando é `g_object_new()`.

 **Observação:** referências indefinidas nem sempre significam que está faltando uma biblioteca. Um dos arquivos-objeto do programa poderia estar ausente no comando de ligação. Geralmente, porém, é fácil diferenciar entre funções de biblioteca e funções de seus arquivos-objeto.

Para corrigir esse problema, você deve inicialmente encontrar a biblioteca gobject e, em seguida, usar a opção `-l` do compilador para fazer a ligação com a biblioteca. Como ocorre com os arquivos de inclusão, as bibliotecas estão espalhadas pelo sistema (`/usr/lib` é o local default do sistema), embora a maioria das bibliotecas esteja em um subdiretório chamado *lib*. No exemplo anterior, o arquivo básico da biblioteca gobject é *libgobject.a*, portanto o nome da biblioteca é gobject. Reunindo tudo, você irá fazer a ligação do programa desta maneira:

```
$ cc -o badobject badobject.o -lgobject
```

Você deve informar o linker sobre as localizações das bibliotecas que não sejam padrão; o parâmetro para isso é `-L`. Vamos supor que o programa badobject exija *libcrud.a* em `/usr/junk/lib`. Para compilar e criar o executável, utilize um comando como este:

```
$ cc -o badobject badobject.o -lgobject -L/usr/junk/lib -lcrud
```

 **Observação:** se quiser procurar uma função em particular em uma biblioteca, utilize o comando `nm`. Esteja preparado para obter muitos dados de saída. Por exemplo, experimente executar: `nm libgobject.a`. (Talvez seja necessário usar o comando `locate` para encontrar *libgobject.a*; muitas distribuições atualmente colocam as bibliotecas em subdiretórios específicos de arquitetura em `/usr/lib`.)

15.1.4 Bibliotecas compartilhadas

Um arquivo de biblioteca que termine com *.a* (como *libgobject.a*) chama-se *biblioteca*

estática. Ao ligar um programa com uma biblioteca estática, o linker copiará o código de máquina do arquivo da biblioteca para o seu executável. Desse modo, o executável final não precisará do arquivo original da biblioteca para executar e, sendo assim, o comportamento do executável jamais mudará.

Entretanto os tamanhos das bibliotecas estão sempre aumentando, assim como o número de bibliotecas em uso, e isso faz com que as bibliotecas estáticas sejam um desperdício em termos de espaço em disco e memória. Além do mais, se descobirmos mais tarde que uma biblioteca estática é inadequada ou que ela não é segura, não haverá nenhuma maneira de alterar qualquer executável que estiver ligado a ela, a não ser que o executável seja recompilado.

As bibliotecas compartilhadas contornam esses problemas. Ao executar um programa ligado a uma dessas bibliotecas, o sistema carregará o código da biblioteca no espaço de memória do processo somente quando for necessário. Muitos processos podem compartilhar o código da mesma biblioteca compartilhada na memória. E se você precisar modificar levemente o código da biblioteca, em geral, isso poderá ser feito sem recompilar nenhum programa.

As bibliotecas compartilhadas têm seus próprios custos: gerenciamento difícil e um procedimento, de certo modo, complicado de ligação. Entretanto você pode manter as bibliotecas compartilhadas sob controle se tiver quatro informações:

- como listar as bibliotecas compartilhadas necessárias a um executável;
- como um executável procura as bibliotecas compartilhadas;
- como ligar um programa com uma biblioteca compartilhada;
- as armadilhas comuns das bibliotecas compartilhadas.

As seções a seguir mostram como usar e manter as bibliotecas compartilhadas de seu sistema. Se você estiver interessado no funcionamento das bibliotecas compartilhadas ou se quiser conhecer os linkers em geral, dê uma olhada no livro *Linkers and Loaders* de John R. Levine (Morgan Kaufmann, 1999), “The Inside Story on Shared Libraries and Dynamic Loading” (A história interna das bibliotecas compartilhadas e da carga dinâmica) de David M. Beazley, Brian D. Ward e Ian R. Cooke (*Computing in Science & Engineering*, setembro/outubro de 2001) ou acesse recursos online como o Program Library HOWTO (<http://dwheeler.com/program-library/>). Também vale a pena ler a página de manual ld.so(8).

Listando dependências de bibliotecas compartilhadas

Os arquivos de bibliotecas compartilhadas normalmente ficam nos mesmos locais em

que estão as bibliotecas estáticas. Os dois diretórios-padrão de biblioteca em um sistema Linux são */lib* e */usr/lib*. O diretório */lib* não deve conter bibliotecas estáticas.

Uma biblioteca compartilhada tem um sufixo que contém *.so* (shared object, ou objeto compartilhado), como em *libc-2.15.so* e em *libc.so.6*. Para ver quais bibliotecas compartilhadas um programa utiliza, execute *ldd prog*, em que *prog* é o nome do executável. A seguir, apresentamos um exemplo para o shell:

```
$ ldd /bin/bash
linux-gate.so.1 => (0xb7799000)
libtinfo.so.5 => /lib/i386-linux-gnu/libtinfo.so.5 (0xb7765000)
libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xb7760000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75b5000)
/lib/ld-linux.so.2 (0xb779a000)
```

Visando a um desempenho otimizado e à flexibilidade, os executáveis por si só geralmente não conhecem as localizações de suas bibliotecas compartilhadas; eles sabem somente os nomes das bibliotecas e, talvez, tenham uma pequena pista sobre o local em que poderão encontrá-las. Um pequeno programa chamado *ld.so* (o *linker/loader dinâmico em tempo de execução*) encontra e carrega as bibliotecas compartilhadas para um programa em tempo de execução. A saída anterior de *ldd* mostra os nomes das bibliotecas à esquerda – é o que o executável conhece. O lado direito mostra os locais em que *ld.so* encontra a biblioteca.

A última linha da saída, nesse caso, mostra a localização propriamente dita de *ld.so*: *ld-linux.so.2*.

Como o *ld.so* encontra as bibliotecas compartilhadas

Um dos pontos comuns de problema para bibliotecas compartilhadas está no fato de o linker dinâmico não conseguir encontrar uma biblioteca. O primeiro local em que o linker dinâmico normalmente *deve* procurar as bibliotecas compartilhadas é em um *path de pesquisa de biblioteca em tempo de execução (rpath)* que esteja pré-configurado para um executável, caso exista. Você verá como criar esse path em breve.

Em seguida, o linker dinâmico olha em uma cache do sistema – */etc/ld.so.cache* – para ver se a biblioteca está em um local-padrão. Essa é uma cache rápida de nomes de arquivos de biblioteca encontrados em diretórios listados no arquivo */etc/ld.so.conf* de configuração de cache.

👉 **Observação:** como é comum em muitos arquivos de configuração do Linux que já vimos, *ld.so.conf* pode incluir vários arquivos em um diretório como */etc/ld.so.conf.d*.

Cada linha de *ld.so.conf* é um diretório que você deseja incluir na cache. A lista de

diretórios geralmente é curta, contendo algo como:

```
/lib/i686-linux-gnu  
/usr/lib/i686-linux-gnu
```

Os diretórios-padrão de bibliotecas – */lib* e */usr/lib* – estão implícitos, o que significa que não é necessário incluí-los em */etc/ld.so.conf*.

Se *ld.so.conf* for alterado ou uma mudança for feita em um dos diretórios de bibliotecas compartilhadas, você deverá recriar o arquivo */etc/ld.so.cache* manualmente com o comando a seguir:

```
# ldconfig -v
```

A opção *-v* disponibiliza informações detalhadas sobre as bibliotecas adicionadas por *ldconfig* à cache e sobre qualquer mudança que for detectada.

Há mais um lugar em que *ld.so* procura bibliotecas compartilhadas: a variável de ambiente *LD_LIBRARY_PATH*. Falaremos disso em breve.

Não adquira o hábito de adicionar itens em */etc/ld.so.conf*. Você deve saber quais bibliotecas compartilhadas estão na cache do sistema; se você colocar os diretórios de todas as pequenas bibliotecas compartilhadas bizarras na cache, haverá o risco de ter conflitos e de ficar com um sistema extremamente desorganizado. Ao compilar softwares que precisem de um path de biblioteca inusitado, forneça um path de pesquisa de biblioteca em tempo de execução ao seu executável. Vamos ver como isso é feito.

Ligando programas a bibliotecas compartilhadas

Vamos supor que você tenha uma biblioteca compartilhada chamada *libweird.so.1* em */opt/obscure/lib*, que deva ser ligada a *myprog*. Faça a ligação do programa da seguinte maneira:

```
$ cc -o myprog myprog.o -Wl,-rpath=/opt/obscure/lib -L/opt/obscure/lib -lweird
```

A opção *-Wl,-rpath* diz ao linker para incluir um diretório que se segue ao path de pesquisa de biblioteca em tempo de execução do executável. Entretanto, mesmo que *-Wl,-rpath* seja usado, a flag *-L* continuará sendo necessária.

Se você tiver um binário preexistente, o programa *patchelf* também poderá ser usado para inserir um path diferente de pesquisa de biblioteca em tempo de execução, porém, em geral, é melhor fazer isso em tempo de compilação.

Problemas com bibliotecas compartilhadas

As bibliotecas compartilhadas oferecem uma flexibilidade notável, sem mencionar alguns hacks realmente incríveis, mas também é possível abusar delas até o ponto em

que seu sistema se transforme em uma confusão completa e absoluta. Três fatos particularmente ruins podem ocorrer:

- bibliotecas ausentes;
- desempenho horrível;
- bibliotecas incompatíveis.

A causa número um de todos os problemas com bibliotecas compartilhadas é a variável de ambiente chamada `LD_LIBRARY_PATH`. Configurar essa variável com um conjunto de nomes de diretórios delimitados com dois-pontos faz o `ld.so` pesquisar os diretórios especificados *antes* de todo o restante ao procurar uma biblioteca compartilhada. É uma maneira ordinária de fazer os programas funcionarem ao mudar a localização de uma biblioteca, se você não tiver o código-fonte do programa e não puder usar `patchelf`, ou se você simplesmente for preguiçoso demais para recompilar os executáveis. Infelizmente, você terá aquilo pelo qual pagar.

Jamais defina `LD_LIBRARY_PATH` em arquivos de inicialização de shell ou quando estiver compilando softwares. Quando o linker dinâmico de tempo de execução encontrar essa variável, com frequência, ele deverá pesquisar todo o conteúdo de cada diretório especificado mais vezes do que você imagina. Isso exerce um impacto enorme no desempenho, porém, mais importante ainda é o fato de que você poderá ter conflitos e bibliotecas incompatíveis, pois o linker de tempo de execução olha esses diretórios para *todos* os programas.

Se for *necessário* usar `LD_LIBRARY_PATH` para executar algum programa ruim, para o qual você não tenha o código-fonte (ou uma aplicação que você prefira não compilar, como o Mozilla ou outra fera), use um script wrapper. Vamos supor que seu executável seja `/opt/crummy/bin/crummy.bin` e que ele precise de algumas bibliotecas compartilhadas que estão em `/opt/crummy/lib`. Crie um script wrapper chamado `crummy` que tenha o seguinte aspecto:

```
#!/bin/sh
LD_LIBRARY_PATH=/opt/crummy/lib
export LD_LIBRARY_PATH
exec /opt/crummy/bin/crummy.bin $@
```

Deixar de usar `LD_LIBRARY_PATH` evita a maioria dos problemas com bibliotecas compartilhadas. Porém outro problema significativo que ocasionalmente surge entre os desenvolvedores é o fato de a API (Application Programming Interface, ou Interface de programação de aplicativos) de uma biblioteca mudar um pouco de uma versão menor para outra, causando problemas em softwares já instalados. As melhores soluções,

nesse caso, são de caráter preventivo: use uma metodologia consistente para instalar bibliotecas compartilhadas usando `-Wl,-rpath` para criar um path de ligação em tempo de execução ou simplesmente use versões estáticas das bibliotecas inusitadas.

15.2 make

Um programa com mais de um arquivo de código-fonte ou que exija opções incomuns de compilação é desajeitado demais para ser compilado manualmente. Esse problema tem estado presente há anos, e o utilitário tradicional de gerenciamento de compilação do Unix que alivia esse problema chama-se `make`. Você deve conhecer um pouco do `make` se estiver executando um sistema Unix, pois os utilitários do sistema às vezes dependem do `make` para funcionar. No entanto este capítulo representa somente a ponta do iceberg. Há livros inteiros sobre o `make`, por exemplo, *Managing Projects with GNU Make* de Robert Mecklenburg (O'Reilly, 2004). Além disso, a maioria dos pacotes Linux é criada usando uma camada adicional em torno do `make` ou de uma ferramenta semelhante. Há vários sistemas de build por aí; daremos uma olhada em um chamado autotools no capítulo 16.

O `make` é um sistema grande, porém não é muito difícil ter uma ideia de como ele funciona. Ao ver um arquivo chamado *Makefile* ou *makefile*, você saberá que está trabalhando com o `make`. (Experimente executar `make` para ver se é possível gerar algo.)

A ideia básica por trás do `make` é o *alvo* (target) – uma meta que você quer alcançar. Um alvo pode ser um arquivo (um arquivo `.o`, um executável e assim por diante) ou um rótulo (label). Além disso, alguns alvos dependem de outros alvos; por exemplo, é preciso ter um conjunto completo de arquivos `.o` antes de poder fazer a ligação de seu executável. Esses requisitos são chamados de *dependências*.

Para gerar um alvo, o `make` segue uma *regra*, por exemplo, uma que especifique como partir de um arquivo-fonte `.c` e chegar a um arquivo-objeto `.o`. O `make` já conhece diversas regras, porém essas regras já existentes podem ser personalizadas para que você crie suas próprias regras.

15.2.1 Um exemplo de Makefile

O Makefile bem simples a seguir gera um programa chamado `myprog` a partir de `aux.c` e de `main.c`:

```
# arquivos-objeto
OBJS=aux.o main.o

all: myprog
```

```
myprog: $(OBJS)
        $(CC) -o myprog $(OBJS)
```

O # na primeira linha desse Makefile indica um comentário.

A próxima linha é somente uma definição de macro; ela define a variável `OBJS` com dois nomes de arquivos-objeto. Isso será importante depois. Por enquanto, preste atenção em como a macro é definida e também em como ela é referenciada posteriormente (`$(OBJS)`).

O próximo item no Makefile contém o primeiro alvo: `all`. O primeiro alvo é sempre o default – o alvo que `make` quer criar quando ele é executado sozinho na linha de comando.

A regra para criar um alvo vem depois dos dois-pontos. Para `all`, esse Makefile diz que você deve satisfazer algo chamado `myprog`. Essa é a primeira dependência no arquivo: `all` depende de `myprog`. Observe que `myprog` pode ser um arquivo ou o alvo de outra regra. Nesse caso, são ambos (a regra para `all` e o alvo de `OBJS`).

Para gerar `myprog`, esse Makefile utiliza a macro `$(OBJS)` nas dependências. A macro é expandida para `aux.o` e `main.o`, portanto `myprog` depende desses dois arquivos (eles devem ser arquivos de verdade, pois não há nenhum alvo com esses nomes em nenhum local no Makefile).

Esse Makefile supõe que você tem dois arquivos-fonte C chamados `aux.c` e `main.c` no mesmo diretório. Executar `make` no Makefile resulta na saída a seguir, que mostra os comandos executados pelo `make`:

```
$ make
cc -c -o aux.o aux.c
cc -c -o main.o main.c
cc -o myprog aux.o main.o
```

A figura 15.1 mostra um diagrama de dependências.

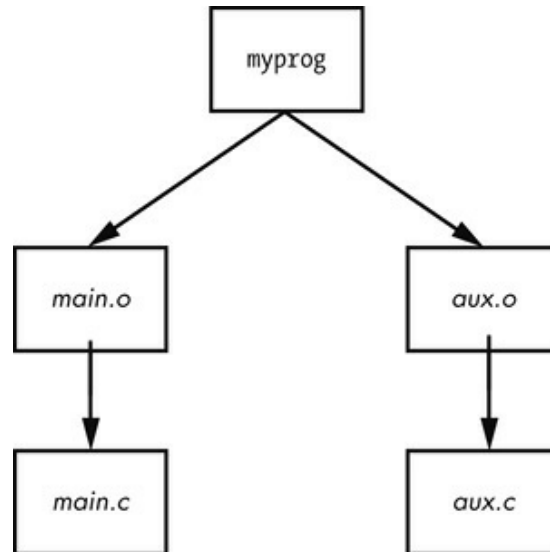


Figura 15.1 – Dependências do Makefile.

15.2.2 Regras prontas

Então como o `make` sabe como ir de `aux.c` para `aux.o`? Afinal de contas, `aux.c` não está no Makefile. A resposta é que o `make` segue suas regras prontas. Ele sabe procurar um arquivo `.c` quando você quer um arquivo `.o` e, além disso, ele sabe como executar `cc -c` nesse arquivo `.c` para atingir seu objetivo de criar um arquivo `.o`.

15.2.3 Geração do programa final

O último passo para chegar a `myprog` é um pouco complicado, porém a ideia é bem clara. Depois que você tiver os dois arquivos-objeto em `$(OBJS)`, o compilador C poderá ser executado de acordo com a linha seguinte (em que `$(CC)` é expandido com o nome do compilador):

```
$(CC) -o myprog $(OBJS)
```

O espaço em branco antes de `$(CC)` é uma tabulação. Você *deve* inserir uma tabulação antes de qualquer comando de verdade em sua própria linha.

Preste atenção em:

```
Makefile:7: *** missing separator. Stop.
```

Um erro como esse significa que o Makefile tem problemas. A tabulação é o separador e, se não houver nenhum separador ou houver outro tipo de interferência, você verá esse erro.

15.2.4 Mantendo-se atualizado

Um último princípio do `make` é que os alvos devem estar atualizados em relação a suas

dependências. Se você digitar `make` duas vezes seguidas para o exemplo anterior, o primeiro comando irá gerar `myprog`, porém o segundo resultará em:

```
make: Nothing to be done for 'all'.
```

Dessa segunda vez, o `make` olhou para suas regras e percebeu que `myprog` já existia, portanto não o gerou novamente, pois nenhuma das dependências havia sofrido alterações desde a última vez que `myprog` foi gerado. Para fazer experiências com isso, faça o seguinte:

1. Execute `touch aux.c`.
2. Execute `make` novamente. Dessa vez, `make` determina que `aux.c` é mais novo que `aux.o` que já está no diretório, portanto `aux.c` será compilado novamente.
3. `myprog` depende de `aux.o` e agora, `aux.o` é mais novo que o `myprog` preexistente, portanto `make` deve criar `myprog` novamente.

Esse tipo de reação em cadeia é bem comum.

15.2.5 Argumentos de linha de comando e opções

Você pode tirar bastante vantagem do `make` se souber como seus argumentos de linha de comando e as opções funcionam.

Uma das opções mais úteis serve para especificar um único alvo na linha de comando. No Makefile anterior, podemos executar `make aux.o` se quisermos somente o arquivo `aux.o`.

Podemos também definir uma macro na linha de comando. Por exemplo, para usar o compilador `clang`, execute:

```
$ make CC=clang
```

Nesse caso, o `make` usa a sua definição de `CC` no lugar de seu compilador `cc` default. As macros de linha de comando são práticas ao testar definições de pré-processador e bibliotecas, especialmente com as macros `CFLAGS` e `LDFLAGS` que serão discutidas em breve.

Na verdade, você não precisa nem mesmo de um Makefile para executar o `make`. Se as regras prontas do `make` corresponderem a um alvo, você poderá simplesmente pedir ao `make` que tente criá-lo. Por exemplo, se você tiver o código-fonte de um programa bem simples chamado `blah.c`, experimente executar `make blah`. A execução de `make` ocorre desta maneira:

```
$ make blah
cc blah.o -o blah
```


Esse uso de `make` funciona somente para os programas C mais elementares; se o seu programa precisar de uma biblioteca ou de algum diretório especial de inclusão, provavelmente você deverá criar um Makefile. Executar `make` sem um Makefile, na verdade, é mais útil quando você estiver lidando com algo como Fortran, Lex ou Yacc e não souber como o compilador ou o utilitário funciona. Por que não deixar que o `make` descubra para você? Mesmo que o `make` falhe em criar o alvo, é provável que ele ainda ofereça uma pista muito boa sobre como usar a ferramenta.

Duas opções de `make` se destacam entre as demais:

- `-n` – exhibe os comandos necessários para um build, porém impede que o `make` realmente execute qualquer comando.
- `-f arquivo` – diz ao `make` para ler de *arquivo* em vez de ler de *Makefile* ou de *makefile*.

15.2.6 Macros e variáveis padrão

O `make` tem várias macros e variáveis especiais. É difícil dizer a diferença entre uma macro e uma variável, portanto usaremos o termo *macro* para nos referirmos a algo que geralmente não muda depois que o `make` começa a gerar os alvos.

Como vimos anteriormente, as macros podem ser definidas no início de seu Makefile. As macros a seguir são as mais comuns:

- `CFLAGS` – são as opções do compilador C. Ao criar código-objeto a partir de um arquivo `.c`, o `make` passa isso como argumento para o compilador.
- `LDFLAGS` – é como `CFLAGS`, porém é usado pelo linker ao criar um executável a partir do código-objeto.
- `LDLIBS` – se você usar `LDFLAGS`, mas não quiser combinar as opções de nomes de biblioteca com o path de pesquisa, coloque as opções de nomes de biblioteca nesse arquivo.
- `CC` – é o compilador C. O default é `cc`.
- `CPPFLAGS` – são as opções do *pré-processador* C. Quando `make` executar o pré-processador C, ele passará a expansão dessa macro como argumento.
- `CXXFLAGS` – o GNU `make` usa isso como flags do compilador C++.

Uma *variável* do `make` muda à medida que os alvos são gerados. Como as variáveis do `make` jamais são definidas manualmente, a lista a seguir inclui o \$.

- `$$` – se estiver em uma regra, isso é expandido com o alvo corrente.
- `$(basename)` – é expandido com o *basename* (nome de base) do alvo corrente. Por exemplo, se

você estiver gerando *blah.o*, isso será expandido para *blah*.

A lista mais abrangente das variáveis do `make` no Linux está no manual `info` do `make`.

👍 **Observação:** tenha em mente que o GNU `make` tem várias extensões, regras prontas e recursos que outras variantes não têm. Não haverá problemas, desde que você esteja executando o Linux, porém se você mudar para um computador Solaris ou BSD e esperar que ele funcione, você poderá ter uma surpresa. No entanto esse é um problema que sistemas de build multiplataforma como o GNU autotools resolvem.

15.2.7 Alvos convencionais

A maioria dos Makefiles contém vários alvos-padrão que realizam tarefas auxiliares relacionadas à compilação.

- `clean` — o alvo `clean` está em toda parte; um `make clean` geralmente instrui o `make` a remover todos os arquivos-objeto e executáveis para que você possa começar tudo de novo ou empacotar o software. A seguir, apresentamos uma regra de exemplo para o Makefile de `myprog`:

`clean:`

```
rm -f $(OBJS) myprog
```

- `distclean` — um Makefile criado por meio do sistema GNU autotools sempre tem um alvo `distclean` para remover tudo o que não fizer parte da distribuição original, incluindo o Makefile. Você verá mais sobre esse assunto no capítulo 16. Em ocasiões muito raras, você poderá perceber que um desenvolvedor optará por não remover o executável com esse alvo, preferindo algo como `realclean` em seu lugar.
- `install` — copia arquivos e programas compilados para o local em que o Makefile acha ser apropriado no sistema. Isso pode ser perigoso, portanto sempre execute um `make -n install` antes para ver o que acontecerá sem realmente executar qualquer comando.
- `test` ou `check` — alguns desenvolvedores fornecem alvos `test` ou `check` para garantir que tudo funcionará após realizar um build.
- `depend` — cria dependências ao chamar o compilador com `-M` para examinar o código-fonte. É um alvo incomum, pois com frequência ele muda o próprio Makefile. Essa não é mais uma prática comum, mas se você se deparar com algumas instruções que digam para usar essa regra, certifique-se de que isso será feito.
- `all` — geralmente, é o primeiro alvo do Makefile. Normalmente, você verá referências a esse alvo no lugar de um executável propriamente dito.

15.2.8 Organizando um Makefile

Apesar de haver vários estilos diferentes de Makefile, a maioria dos programadores se

atém a algumas regras gerais. Para mencionar uma, na primeira parte do Makefile (nas definições de macro), você deverá ver bibliotecas e includes agrupados de acordo com o pacote:

```
MYPACKAGE_INCLUDES=-I/usr/local/include/mypackage
MYPACKAGE_LIB=-L/usr/local/lib/mypackage -lmypackage
PNG_INCLUDES=-I/usr/local/include
PNG_LIB=-L/usr/local/lib -lpng
```

Cada tipo de flag de compilador e de linker com frequência recebe uma macro como estas:

```
CFLAGS=$(CFLAGS) $(MYPACKAGE_INCLUDES) $(PNG_INCLUDES)
LDFLAGS=$(LDFLAGS) $(MYPACKAGE_LIB) $(PNG_LIB)
```

Os arquivos-objeto normalmente são agrupados de acordo com os executáveis. Por exemplo, suponha que você tenha um pacote que crie executáveis chamados *boring* e *trite*. Cada qual tem seu próprio arquivo-fonte *.c* e exige o código que está em *util.c*. Você poderá ver algo como:

```
UTIL_OBJS=util.o
BORING_OBJS=$(UTIL_OBJS) boring.o
TRITE_OBJS=$(UTIL_OBJS) trite.o
PROGS=boring trite
```

O restante do Makefile poderá ter o seguinte aspecto:

```
all: $(PROGS)

boring: $(BORING_OBJS)
    $(CC) -o $@ $(BORING_OBJS) $(LDFLAGS)

trite: $(TRITE_OBJS)
    $(CC) -o $@ $(TRITE_OBJS) $(LDFLAGS)
```

Você pode combinar os dois alvos executáveis em uma só regra, mas geralmente essa não é uma boa ideia, pois você não poderá mover facilmente uma regra para outro Makefile, apagar um executável ou agrupar os executáveis de modo diferente. Além do mais, as dependências estariam incorretas: se você tivesse somente uma regra para *boring* e *trite*, *trite* dependeria de *boring.c*, *boring* dependeria de *trite.c* e o *make* sempre tentaria recriar ambos os programas sempre que um dos dois arquivos-fonte fosse alterado.

 **Observação:** se for necessário definir uma regra especial para um arquivo-objeto, coloque a regra para esse arquivo imediatamente antes da regra que gera o executável. Se vários executáveis usarem o mesmo arquivo-objeto, coloque a regra para o objeto acima de todas as regras de executáveis.

15.3 Debuggers

O debugger-padrão nos sistemas Linux é o `gdb`; frontends amigáveis ao usuário como o Eclipse IDE e os sistemas Emacs também estão disponíveis. Para permitir um debugging completo de seus programas, execute o compilador com `-g` para gravar uma tabela de símbolos e outras informações de debugging no executável. Para iniciar o `gdb` em um executável chamado *programa*, execute:

```
$ gdb programa
```

Você deverá obter um prompt (`gdb`). Para executar *programa* com o argumento *opções* de linha de comando, digite isto no prompt (`gdb`):

```
(gdb) run opções
```

Se o programa funcionar, ele deverá iniciar, executar e sair normalmente. Entretanto, se houver um problema, o `gdb` vai parar, exibir o código-fonte com falha e fazer você retornar ao prompt (`gdb`). Como o fragmento de código-fonte geralmente dá uma pista do problema, provavelmente você irá querer exibir o valor de uma determinada variável com a qual o problema possa estar relacionado. (O comando `print` também funciona para arrays e estruturas em C.)

```
(gdb) print variável
```

Para fazer o `gdb` parar o programa em qualquer ponto do código-fonte original, utilize o recurso de breakpoint. No comando a seguir, *arquivo* é um arquivo de código-fonte e *num_linha* é o número da linha desse arquivo em que o `gdb` deve parar:

```
(gdb) break arquivo:num_linha
```

Para dizer ao `gdb` que continue a executar o programa, utilize:

```
(gdb) continue
```

Para limpar um breakpoint, digite:

```
(gdb) clear arquivo:num_linha
```

Esta seção forneceu somente as mais breves introduções ao `gdb`; o `gdb` inclui um manual extenso que pode ser lido online, impresso ou você pode adquirir o livro *Debugging with GDB*, 10ª edição, de Richard M. Stallman et al. (GNU Press, 2011). Outro guia para debugging é o livro *The Art of Debugging* de Norman Matloff e Peter Jay Salzman (No Starch Press, 2008).

👍 **Observação:** se você estiver interessado em identificar problemas de memória e executar testes de perfil, experimente usar o Valgrind (<http://valgrind.org/>).

15.4 Lex e Yacc

Você poderá se deparar com o Lex e o Yacc ao compilar programas que leem arquivos

de configuração ou de comandos. Essas ferramentas são blocos de construção para linguagens de programação.

- O Lex é um *gerador de tokens* (tokenizer) que transforma texto em tags numeradas com rótulos. A versão GNU/Linux chama-se *flex*. Você poderá precisar de uma flag de linker `-ll` ou `-lfl` em conjunto com o Lex.
- O Yacc é um *parser* que tenta ler tokens de acordo com uma *gramática*. O GNU parser é o *bison*; para ter compatibilidade com o Yacc, execute `bison -y`. Você poderá precisar da flag de linker `-ly`.

15.5 Linguagens de scripting

Tempos atrás, o administrador médio de sistemas Unix não precisava se preocupar muito com linguagens de scripting além do Bourne shell e do awk. Os shell scripts (discutidos no capítulo 11) continuam sendo uma parte importante do Unix, porém o awk, de certo modo, tem desaparecido da área de scripting. Entretanto muitos sucessores capacitados surgiram, e vários programas de sistema, na realidade, passaram de C para linguagens de scripting (por exemplo, a versão razoável do programa `whois`). Vamos dar uma olhada no básico sobre scripting.

A primeira informação que você deve ter sobre qualquer linguagem de scripting é que a primeira linha de um script se parece com o shebang de um Bourne shell script. Por exemplo, um script Python começa com algo como:

```
#!/usr/bin/python
```

Ou com:

```
#!/usr/bin/env python
```

No Unix, *qualquer* arquivo-texto executável que comece com `#!` é um script. O nome do path que se segue a esse prefixo é o executável do interpretador da linguagem de scripting. Quando o Unix tenta executar um arquivo que comece com um shebang `#!`, o programa que se segue a `#!` será executado, com o restante do arquivo como a entrada-padrão. Desse modo, até mesmo o que está sendo mostrado a seguir é um script:

```
#!/usr/bin/tail -2
```

```
This program won't print this line,  
but it will print this line...  
and this line, too.
```

A primeira linha de um shell script normalmente contém um dos problemas mais comuns e básicos de scripts: um path inválido para o interpretador da linguagem de scripting. Por exemplo, suponha que você tenha chamado o script anterior de `myscript`. E

se `tail` na realidade estivesse em `/bin` e não em `/usr/bin` em seu sistema? Nesse caso, executar `myscript` deverá gerar o seguinte erro:

```
bash: ./myscript: /usr/bin/tail: bad interpreter: No such file or directory
```

Não espere que mais de um argumento na primeira linha do script vá funcionar. Ou seja, o `-2` no exemplo anterior poderá funcionar, porém, se você adicionar outro argumento, o sistema poderá decidir tratar o `-2` e o novo argumento como um argumento maior, com espaço e tudo. Isso pode variar de sistema para sistema; não teste sua paciência em algo tão insignificante quanto isso.

Vamos agora dar uma olhada em algumas linguagens existentes por aí.

15.5.1 Python

O Python é uma linguagem de scripting com muitos seguidores e um conjunto eficaz de recursos, por exemplo, processamento de texto, acesso a banco de dados, rede e multithreading. Ele tem um modo interativo eficiente e um modelo de objetos muito organizado.

O executável do Python é o `python`, que normalmente está em `/usr/bin`. Entretanto o Python não é usado somente a partir da linha de comando para scripts. Um lugar em que você o encontrará é como uma ferramenta de criação de sites. O livro *Python Essential Reference*, 4ª edição, de David M. Beazley (Addison-Wesley, 2009), é uma ótima referência e contém um pequeno tutorial no início para que você possa dar os primeiros passos.

15.5.2 Perl

Uma das linguagens de scripting mais antigas de terceiros do Unix é o Perl. É o “canivete suíço” original das ferramentas de programação. Embora o Perl tenha perdido bastante terreno para o Python nos últimos anos, ele é particularmente excelente no processamento de texto, em conversão e manipulação de arquivos, e você poderá encontrar várias ferramentas criadas com ele. O livro *Learning Perl*, 6ª edição, de Randal L. Schwartz, brian d foy e Tom Phoenix (O'Reilly, 2011), é uma introdução em estilo de tutorial; uma referência mais extensa é *Modern Perl* de Chromatic (Onyx Neon Press, 2014).

15.5.3 Outras linguagens de scripting

As seguintes linguagens de scripting também poderão ser encontradas:

- PHP – é uma linguagem de processamento de hipertexto, geralmente encontrada em

scripts web dinâmicos. Algumas pessoas usam o PHP para scripts autônomos. O site do PHP está em <http://www.php.net/>.

- Ruby – os fanáticos por orientação a objetos e muitos desenvolvedores web gostam de programar com essa linguagem (<http://www.ruby-lang.org/>).
- JavaScript – essa linguagem é usada em navegadores web, principalmente para manipular conteúdo dinâmico. A maioria dos programadores experientes a rejeita como uma linguagem de scripting independente por causa de seus vários defeitos, porém é quase impossível evitá-la ao trabalhar com programação web. Você poderá encontrar uma implementação chamada Node.js, com um executável chamado `node` em seu sistema.
- Emacs Lisp – é uma variedade da linguagem de programação Lisp usada pelo editor de texto Emacs.
- Matlab, Octave – o Matlab é uma linguagem de programação e uma biblioteca comerciais para matrizes e operações matemáticas. Há um projeto de software gratuito muito semelhante chamado Octave.
- R – é uma linguagem gratuita e popular de análise estatística. Acesse o site <http://www.r-project.org/> e dê uma olhada no livro *The Art of R Programming* de Norman Matloff (No Starch Press, 2011) para obter mais informações.
- Mathematica – é outra linguagem de programação matemática comercial com bibliotecas.
- m4 – é uma linguagem de processamento de macros normalmente encontrada somente no GNU autotools.
- Tcl – o Tcl (Tool command language) é uma linguagem de scripting simples, geralmente associada ao kit de ferramentas de interface gráfica de usuário Tk e ao Expect, que é um utilitário para automação. Embora o Tcl não goze do uso amplo que já teve um dia, não menospreze a sua eficácia. Muitos desenvolvedores veteranos preferem o Tk, especialmente pelas funcionalidades incluídas. Acesse <http://www.tcl.tk/> para obter mais informações sobre o Tk.

15.6 Java

O Java é uma linguagem compilada como o C, com uma sintaxe mais simples e um suporte eficaz para programação orientada a objetos. Ele tem alguns nichos nos sistemas Unix. Para mencionar um, com frequência, ele é usado como ambiente de aplicações web, e é popular para aplicações especializadas. Por exemplo, as

aplicações Android normalmente são escritas em Java. Apesar de não ser visto com frequência em um desktop Linux comum, você deve saber como o Java funciona, pelo menos para aplicações autônomas.

Há dois tipos de compiladores Java: compiladores nativos para gerar código de máquina para o seu sistema (como um compilador C) e compiladores de bytecode para serem usados por um interpretador de bytecodes [às vezes chamado de *máquina virtual*, que é diferente da máquina virtual oferecida por um hipervisor (hypervisor), conforme descrito no capítulo 17]. Você praticamente sempre encontrará bytecodes no Linux.

Os arquivos de bytecode Java terminam com *.class*. O JRE (Java Runtime Environment) contém todos os programas necessários para executar bytecodes Java. Para executar um arquivo de bytecodes, utilize:

```
$ java arquivo.class
```

Você também poderá encontrar arquivos de bytecodes que terminem com *.jar*, que são coleções de arquivos *.class* reunidos. Para executar um arquivo *.jar*, utilize a sintaxe a seguir:

```
$ java -jar arquivo.jar
```

Às vezes, você deverá definir a variável de ambiente `JAVA_HOME` com o prefixo de sua instalação Java. Se você realmente estiver com azar, poderá ser preciso usar `CLASSPATH` para incluir todos os diretórios contendo as classes esperadas pelo seu programa. Esse é um conjunto de diretórios delimitados por dois-pontos, como a variável `PATH` normal para executáveis.

Se você precisar compilar um arquivo *.java* e gerar bytecodes, será necessário ter o JDK (Java Development Kit). O compilador `javac` do JDK pode ser executado para criar alguns arquivos *.class*:

```
$ javac arquivo.java
```

O JDK também vem com o `jar` — um programa que pode criar e separar arquivos *.jar*. Ele funciona como o `tar`.

15.7 Próximos passos: compilando pacotes

O mundo dos compiladores e das linguagens de scripting é vasto e está constantemente em expansão. Na época desta publicação, novas linguagens compiladas como o Go (golang) e o Swift estavam ganhando popularidade.

O conjunto de infraestrutura de compiladores LLVM (<http://llvm.org/>) facilitou

significativamente o desenvolvimento de compiladores. Se você estiver interessado em fazer o design e implementar um compilador, dois bons livros são: *Compiladores: princípios, técnicas e ferramentas*, 2ª edição, de Alfred V. Aho et al. (Pearson, 2008), e *Modern Compiler Design*, 2ª edição, de Dick Grune et al. (Springer, 2012). Para desenvolvimento com linguagens de scripting, geralmente é melhor procurar recursos online, pois as implementações variam muito.

Agora que você já conhece o básico sobre as ferramentas de programação do sistema, está pronto para ver o que elas podem fazer. O próximo capítulo trata da geração de pacotes no Linux a partir do código-fonte.

CAPÍTULO 16

Introdução à compilação de software a partir de código-fonte C

A maioria dos pacotes de software Unix de terceiros que não são proprietários se apresenta na forma de código-fonte que pode ser compilado e instalado. Um motivo para isso é que o Unix (e o próprio Linux) tem tantas variantes e arquiteturas diferentes que seria difícil distribuir pacotes binários para todas as combinações possíveis de plataforma. O outro motivo, que é no mínimo tão importante quanto esse, é que uma distribuição ampla de código-fonte por meio da comunidade Unix incentiva os usuários a contribuírem com correções de bug e com novos recursos para o software, dando sentido ao termo *código aberto*.

Você pode obter quase tudo que vir em um sistema Linux na forma de código-fonte – desde o kernel e a biblioteca C até os navegadores web. É até mesmo possível atualizar e expandir todo o seu sistema ao (re)instalar partes dele a partir do código-fonte. Entretanto você não *deve* atualizar seu computador instalando *tudo* a partir do código-fonte, a menos que realmente goste do processo ou tenha outros motivos.

As distribuições Linux normalmente oferecem maneiras mais simples de atualizar as partes essenciais do sistema, por exemplo, os programas em */bin*, e uma propriedade particularmente importante das distribuições é que elas normalmente corrigem problemas de segurança bem rapidamente. Mas não espere que sua distribuição disponibilize tudo para você. Eis algumas razões pelas quais você pode querer instalar determinados pacotes por conta própria:

- Controlar opções de configuração.
- Instalar o software no local em que você quiser. É possível até mesmo instalar várias versões diferentes do mesmo pacote.
- Controlar a versão a ser instalada. As distribuições nem sempre permanecem atualizadas com as versões mais recentes de todos os pacotes, particularmente com os add-ons aos pacotes de software (por exemplo, as bibliotecas Python).
- Entender melhor como um pacote funciona.

16.1 Sistemas de geração de software

Há vários ambientes de programação no Linux, desde o C tradicional até linguagens de scripting interpretadas como o Python. Cada um deles normalmente tem pelo menos um sistema distinto para geração e instalação de pacotes, além das ferramentas que uma distribuição Linux oferece.

Daremos uma olhada na compilação e na instalação de código-fonte C neste capítulo, usando somente um desses sistemas – os scripts de configuração gerados pelo pacote GNU autotools. Esse sistema geralmente é considerado estável, e muitos dos utilitários básicos do Linux o utiliza. Como ele é baseado em ferramentas como o `make`, depois de vê-lo em ação, você poderá aplicar seu conhecimento em outros sistemas de build.

Instalar um pacote a partir de código-fonte C geralmente envolve os passos a seguir:

1. Desempacotar o arquivo de código-fonte.
2. Configurar o pacote.
3. Executar `make` para gerar os programas.
4. Executar `make install` ou um comando de instalação específico da distribuição para instalar o pacote.

👍 **Observação:** você deve entender o básico que está no capítulo 15 antes de prosseguir com este capítulo.

16.2 Desempacotando pacotes com código-fonte C

Uma distribuição de pacote com código-fonte normalmente se apresenta na forma de um arquivo `.tar.gz`, `.tar.bz2` ou `.tar.xz`, e você deve desempacotar o arquivo, conforme descrito na seção 2.18. Antes de desempacotá-lo, porém, verifique o conteúdo do arquivo usando `tar tvf` ou `tar ztvf`, pois alguns pacotes não criam seus próprios subdiretórios no diretório em que a extração for feita.

Uma saída como a que se segue significa que provavelmente não há problemas e que o pacote poderá ser desempacotado:

```
package-1.23/Makefile.in
package-1.23/README
package-1.23/main.c
package-1.23/bar.c
--trecho omitido--
```

No entanto você poderá perceber que nem todos os arquivos estão em um diretório comum (como `package-1.23` no exemplo anterior):

```
Makefile
README
main.c
--trecho omitido--
```

Fazer a extração de um arquivo como esse pode provocar uma enorme confusão em seu diretório corrente. Para evitar isso, crie um novo diretório e acesse-o com `cd` antes de extrair o conteúdo do arquivo.

Por fim, tome cuidado com pacotes que contenham arquivos com nomes de path absolutos como estes:

```
/etc/passwd
/etc/inetd.conf
```

Provavelmente, você não irá se deparar com nada desse tipo, mas, se isso ocorrer, remova o arquivo de seu sistema. É provável que ele contenha um cavalo de Troia ou outros códigos maliciosos.

16.2.1 Por onde começar

Depois de ter extraído o conteúdo de um arquivo com códigos-fonte e ter um conjunto de arquivos diante de você, procure ter uma noção do pacote. Em particular, procure os arquivos *README* e *INSTALL*. Sempre dê uma olhada em qualquer arquivo *README* antes, pois geralmente ele contém uma descrição do pacote, um pequeno manual, dicas de instalação e outras informações úteis. Muitos pacotes também vêm com arquivos *INSTALL* contendo instruções sobre como compilar e instalar o pacote. Preste atenção, em particular, em opções especiais do compilador e em definições.

Além dos arquivos *README* e *INSTALL*, você encontrará outros arquivos no pacote que, de modo geral, se enquadrarão nas três classes a seguir:

- Arquivos relacionados ao sistema `make`, como *Makefile*, *Makefile.in*, *configure* e *CMakeLists.txt*. Alguns pacotes bem antigos vêm com um *Makefile* que poderá ser preciso modificar, porém a maioria usa um utilitário para configuração como o GNU `autoconf` ou o `CMake`. Esses pacotes vêm com um script ou com um arquivo de configuração (como *configure* ou *CMakeLists.txt*) para ajudar a gerar um *Makefile* a partir de *Makefile.in*, de acordo com as configurações de seu sistema e as opções de configuração.
- Arquivos de código-fonte que terminem com *.c*, *.h* ou *.cc*. Os arquivos de código-fonte C podem aparecer praticamente em qualquer ponto de um diretório de pacote. Arquivos com código-fonte C++ normalmente têm sufixos *.cc*, *.C* ou *.cxx*.
- Arquivos-objeto que terminem com *.o* ou binários. Normalmente, não haverá nenhum

arquivo-objeto nas distribuições de código-fonte, porém você poderá encontrar alguns nos raros casos em que o mantenedor do pacote não tem permissão para disponibilizar determinados códigos-fonte e seja necessário fazer algo especial para usar os arquivos-objeto. Na maioria dos casos, os arquivos-objeto (ou os executáveis binários) em uma distribuição de código-fonte significam que o pacote não foi composto de maneira adequada e que você deverá executar `make clean` para garantir que uma nova compilação seja feita.

16.3 GNU autoconf

Apesar de o código-fonte C geralmente ser bem portátil, as diferenças em cada plataforma fazem com que seja impossível compilar a maioria dos pacotes com um único Makefile. As primeiras soluções para esse problema consistiam em disponibilizar Makefiles individuais para cada sistema operacional ou um Makefile que fosse fácil de modificar. Essa abordagem evoluiu para o uso de scripts que geram Makefiles de acordo com uma análise do sistema usado para gerar o pacote.

O GNU autoconf é um sistema popular para geração automática de Makefile. Os pacotes que usam esse sistema vêm com arquivos de nome *configure*, *Makefile.in* e *config.h.in*. Os arquivos *.in* são templates; a ideia é executar o script `configure` para descobrir as características de seu sistema e, em seguida, fazer substituições nos arquivos *.in* para criar os verdadeiros arquivos de build. Para o usuário final, é fácil; para gerar um Makefile a partir de *Makefile.in*, execute `configure`:

```
$ ./configure
```

Você deverá obter vários dados de diagnóstico na saída à medida que o script verificar seu sistema em busca dos pré-requisitos. Se tudo sair bem, `configure` criará um ou mais Makefiles e um arquivo *config.h* bem como um arquivo de cache (*config.cache*) para que não seja necessário executar determinados testes novamente.

Agora você poderá executar `make` para compilar o pacote. Um passo bem-sucedido na execução de `configure` não significa necessariamente que o passo `make` funcionará, porém as chances são muito boas. (Veja a seção 16.6 para resolver problemas quando houver falha no `configure` e na compilação.)

Vamos fazer alguns experimentos iniciais com o processo.

 **Observação:** nesse ponto, você deverá ter todas as ferramentas de build necessárias disponíveis em seu sistema. Para Debian e Ubuntu, a maneira mais fácil é instalar o pacote `build-essential`; nos sistemas do tipo Fedora, use o `groupinstall` “Development Tools” (Ferramentas de desenvolvimento).

16.3.1 Um exemplo de autoconf

Antes de discutir o modo como podemos alterar o comportamento do autoconf, vamos dar uma olhada em um exemplo simples para que você saiba o que podemos esperar. Você irá instalar o pacote GNU coreutils em seu próprio diretório home (para garantir que você não criará confusão em seu sistema). Obtenha o pacote a partir de <http://ftp.gnu.org/gnu/coreutils/> (a versão mais recente geralmente é a melhor), descompacte-a, acesse seu diretório e configure-o da seguinte maneira:

```
$ ./configure --prefix=$HOME/mycoreutils
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
--trecho omitido--
config.status: executing po-directories commands
config.status: creating po/POTFILES
config.status: creating po/Makefile
```

Agora execute o make:

```
$ make
GEN    lib/alloca.h
GEN    lib/c++defs.h
--trecho omitido--
make[2]: Leaving directory '/home/juser/coreutils-8.22/gnulib-tests'
make[1]: Leaving directory '/home/juser/coreutils-8.22'
```

Em seguida, experimente executar um dos arquivos executáveis que acabamos de criar, por exemplo, `./src/ls`, e experimente executar `make check` para realizar uma série de testes no pacote. (Isso pode demorar um pouco, mas é interessante de ver.)

Por fim, você estará pronto para instalar o pacote. Faça um “dry run” com `make -n` antes para ver o que `make install` fará, sem realmente realizar a instalação:

```
$ make -n install
```

Navegue pela saída e, se nada parecer estranho (como instalar em outro local que não seja em seu diretório *mycoreutils*), faça a instalação propriamente dita:

```
$ make install
```

Agora você deverá ter um subdiretório chamado *mycoreutils* em seu diretório home contendo *bin*, *share* e outros subdiretórios. Dê uma olhada em alguns dos programas em *bin* (você acabou de gerar muitas das ferramentas básicas que conhecemos no capítulo 2). Por fim, como o diretório *mycoreutils* foi configurado para ser independente do restante de seu sistema, você poderá removê-lo totalmente, sem se preocupar em causar danos.

16.3.2 Fazendo a instalação com uma ferramenta de empacotamento

Na maioria das distribuições, é possível instalar novos softwares na forma de um pacote que você poderá manter posteriormente com as ferramentas de empacotamento de sua distribuição. As distribuições baseadas em Debian como o Ubuntu talvez sejam as mais simples; em vez de executar um `make install` simples, isso poderá ser feito com o utilitário `checkinstall` da seguinte maneira:

```
# checkinstall make install
```

Use a opção `--pkgname=nome` para dar um nome específico ao seu novo pacote.

Criar um pacote RPM é um pouco mais complicado, pois você deverá criar uma árvore de diretório antes para o(s) seu(s) pacote(s). Isso pode ser feito com o comando `rpmdev-setuptree`; quando isso for concluído, o utilitário `rpmbuild` poderá ser usado para executar o restante dos passos. É melhor seguir um tutorial online para esse processo.

16.3.3 Opções do script configure

Você acabou de ver uma das opções mais úteis do script `configure`: usar `--prefix` para especificar o diretório de instalação. Por padrão, o alvo `install` de um Makefile gerado por `autoconf` utiliza um *prefixo* igual a `/usr/local`, ou seja, os programas binários ficam em `/usr/local/bin`, as bibliotecas em `/usr/local/lib` e assim por diante.

Com frequência, você vai querer mudar esse prefixo desta maneira:

```
$ ./configure --prefix=novo_prefixo
```

A maioria das versões de `configure` tem uma opção `--help` que lista outras opções de configuração. Infelizmente, a lista em geral é tão longa que às vezes é difícil determinar o que pode ser importante, por isso apresentamos, a seguir, algumas opções essenciais:

- `--bindir=diretório` — instala os executáveis em *diretório*.
- `--sbindir=diretório` — instala os executáveis do sistema em *diretório*.
- `--libdir=diretório` — instala bibliotecas em *diretório*.
- `--disable-shared` — impede que o pacote gere bibliotecas compartilhadas. Conforme a biblioteca, isso pode evitar dores de cabeça mais tarde (veja a seção 15.1.4).
- `--with-pacote=diretório` — diz ao `configure` que *pacote* está em *diretório*. Isso é prático quando uma biblioteca necessária estiver em um local que não seja padrão. Infelizmente, nem todos os scripts `configure` reconhecem esse tipo de opção, e pode ser difícil determinar a sintaxe exata.

Usando diretórios separados de build

Você pode criar diretórios separados de build se quiser fazer experiências com algumas dessas opções. Para isso, crie um novo diretório em outro local no sistema e, a partir desse diretório, execute o script `configure` no diretório de código-fonte original do pacote. Você perceberá que `configure` criará um farm de links simbólicos em seu novo diretório de build, em que todos os links apontarão de volta para a árvore de códigos-fonte no diretório original do pacote. (Alguns desenvolvedores preferem gerar seus pacotes dessa maneira, pois a árvore original de códigos-fonte jamais será modificada. Isso também será útil se você quiser gerar o pacote para mais de uma plataforma ou um conjunto de opções de configuração usando o mesmo pacote de códigos-fonte.)

16.3.4 Variáveis de ambiente

Você pode influenciar `configure` com variáveis de ambiente que o script `configure` colocará nas variáveis de `make`. As mais importantes são `CPPFLAGS`, `CFLAGS` e `LDFLAGS`. Porém esteja ciente de que `configure` pode ser bem exigente no que diz respeito às variáveis de ambiente. Por exemplo, normalmente você deverá usar `CPPFLAGS` em vez de `CFLAGS` para diretórios de arquivos de cabeçalho, pois `configure` geralmente executa o pré-processador de modo independente do compilador.

No `bash`, a maneira mais fácil de enviar uma variável de ambiente para o `configure` é colocando a atribuição da variável na frente de `./configure` na linha de comando. Por exemplo, para definir uma macro `DEBUG` para o pré-processador, utilize o comando a seguir:

```
$ CPPFLAGS=-DDEBUG ./configure
```

👍 **Observação:** você também pode passar uma variável como uma opção de `configure`; por exemplo:

```
$ ./configure CPPFLAGS=-DDEBUG
```

As variáveis de ambiente são especialmente práticas quando `configure` não sabe em que locais deve procurar arquivos de inclusão e bibliotecas de terceiros. Por exemplo, para fazer o pré-processador pesquisar em `dir_include`, execute o comando a seguir:

```
$ CPPFLAGS=-Idir_include ./configure
```

Como vimos na seção 15.2.6, para fazer o linker olhar em `dir_lib`, utilize o comando a seguir:

```
$ LDFLAGS=-Ldir_lib ./configure
```

Se `dir_lib` tiver bibliotecas compartilhadas (veja a seção 15.1.4), o comando anterior provavelmente não definirá o path do linker dinâmico de tempo de execução. Nesse caso, utilize a opção `-rpath` do linker, além de `-L`:

```
$ LDFLAGS="-Ldir_lib -Wl,-rpath=dir_lib" ./configure
```


Tome cuidado ao definir variáveis. Um pequeno erro pode causar erros no compilador e fazer o `configure` falhar. Por exemplo, suponha que você tenha se esquecido do `-I`, como mostrado aqui:

```
$ CPPFLAGS=Idir_include ./configure
```

Isso resulta em um erro como o que se segue:

```
configure: error: C compiler cannot create executables
See 'config.log' for more details
```

Ao explorar o *config.log* gerado a partir dessa tentativa com falha, temos:

```
configure:5037: checking whether the C compiler works
configure:5059: gcc Idir_include conftest.c >&5
gcc: error: Idir_include: No such file or directory
configure:5063: $? = 1
configure:5101: result: no
```

16.3.5 Alvos do autoconf

Depois que o `configure` estiver funcionando, você perceberá que o Makefile que ele gera tem vários outros alvos úteis, além de `all` e `install`, que são padrões:

- `make clean` — como descrito no capítulo 15, isso remove todos os arquivos-objeto, executáveis e bibliotecas.
- `make distclean` — é semelhante a `make clean`, exceto pelo fato de remover todos os arquivos gerados automaticamente, incluindo Makefiles, *config.h*, *config.log* e assim por diante. A ideia é que a árvore de códigos-fonte pareça ser uma distribuição recém-desempacotada após a execução de `make distclean`.
- `make check` — alguns pacotes vêm com uma bateria de testes para verificar se os programas compilados funcionam adequadamente; o comando `make check` executa os testes.
- `make install-strip` — é como `make install`, exceto pelo fato de remover a tabela de símbolos e outras informações de debugging dos executáveis e das bibliotecas ao fazer a instalação. Binários em que essas informações foram removidas ocupam muito menos espaço.

16.3.6 Arquivos de log do autoconf

Se algo der errado durante o processo do `configure` e a causa não for óbvia, você poderá analisar *config.log* para identificar o problema. Infelizmente, *config.log* geralmente é um arquivo gigantesco, o que pode fazer com que seja difícil localizar a origem exata

do problema.

A abordagem geral para encontrar o problema consiste em ir até o final de *config.log* (por exemplo, teclando G em *less*) e, em seguida, retornar pelas páginas até que seja possível vê-lo. Entretanto ainda há muitos dados no final, pois o *configure* faz o dump de todo o seu ambiente aí, incluindo variáveis de saída, variáveis de cache e outras definições. Portanto, em vez de ir até o final e retroceder pelas páginas, vá para o final e faça uma pesquisa reversa em busca de uma string como *for more details* ou procure outra parte próxima ao final da saída com falha de *configure*. (Lembre-se de que você pode iniciar uma pesquisa reversa em *less* com o comando *?*.) Há uma boa chance de que o erro esteja imediatamente acima daquilo que você encontrar em sua pesquisa.

16.3.7 pkg-config

Há tantas bibliotecas de terceiros que manter todas elas em um local comum pode ser confuso. Contudo instalar cada uma com um prefixo diferente pode resultar em problemas ao gerar pacotes que exijam essas bibliotecas de terceiros. Por exemplo, se você quiser compilar o OpenSSH, será preciso ter a biblioteca OpenSSL. Como você informa a localização das bibliotecas OpenSSL e quais bibliotecas são necessárias ao processo de configuração do OpenSSH?

Muitas bibliotecas atualmente usam o programa *pkg-config* não só para divulgar as localizações de seus arquivos de inclusão e as bibliotecas, mas também para especificar as flags exatas necessárias para compilar e ligar um programa. A sintaxe é a seguinte:

```
$ pkg-config opções pacote1 pacote2 ...
```

Por exemplo, para encontrar as bibliotecas exigidas pelo OpenSSL, o comando a seguir pode ser executado:

```
$ pkg-config --libs openssl
```

A saída deverá ser algo como:

```
-lssl -lcrypto
```

Para ver todas as bibliotecas conhecidas pelo *pkg-config*, execute o comando a seguir:

```
$ pkg-config --list-all
```

Como o pkg-config funciona

Se você espiar por trás das cortinas, verá que o *pkg-config* encontra informações de pacotes ao ler os arquivos de configuração que terminem com *.pc*. Por exemplo, a seguir, apresentamos o *openssl.pc* para a biblioteca de socket do OpenSSL, como visto

em um sistema Ubuntu (localizado em */usr/lib/i386-linux-gnu/pkgconfig*):

```
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib/i386-linux-gnu
includedir=${prefix}/include

Name: OpenSSL
Description: Secure Sockets Layer and cryptography libraries and tools
Version: 1.0.1
Requires:
Libs: -L${libdir} -lssl -lcrypto
Libs.private: -ldl -lz
Cflags: -I${includedir} exec_prefix=${prefix}
```

Esse arquivo pode ser alterado, por exemplo, ao adicionar `-Wl,-rpath=${libdir}` às flags de biblioteca de modo a definir um path de linker dinâmico em tempo de execução. No entanto a pergunta mais importante é como o `pkg-config` encontra os arquivos `.pc`, antes de tudo. Por padrão, o `pkg-config` olha o diretório *lib/pkgconfig* de seu prefixo de instalação. Por exemplo, um `pkg-config` instalado com um prefixo */usr/local* olha em */usr/local/lib/pkgconfig*.

Instalando arquivos de `pkg-config` em locais que não são padrão

Infelizmente, por padrão, o `pkg-config` não lê nenhum arquivo `.pc` fora de seu prefixo de instalação. Portanto um arquivo `.pc` que esteja em um local que não seja padrão, como */opt/openssl/lib/pkgconfig/openssl.pc*, estará fora de alcance de qualquer instalação de `pkg-config`. Há duas maneiras básicas de disponibilizar arquivos `.pc` fora do prefixo de instalação de `pkg-config`:

- Criar links simbólicos (ou cópias) dos arquivos `.pc` para o diretório *pkgconfig* central.
- Definir sua variável de ambiente `PKG_CONFIG_PATH` para que inclua qualquer diretório *pkgconfig* extra. Essa estratégia não funciona bem para todo o sistema.

16.4 Efetuando instalações

Saber *como* gerar e instalar um software é bom, porém saber *quando* e *onde* instalar seus próprios pacotes é mais útil ainda. As distribuições Linux tentam colocar o máximo possível de softwares na instalação, e você sempre deve verificar se será melhor instalar um pacote por conta própria. Eis as vantagens de você mesmo fazer instalações:

- É possível personalizar os defaults dos pacotes.
- Ao instalar um pacote, com frequência, você terá uma imagem mais clara de como usá-lo.
- Você controla a versão a ser executada.
- É mais fácil fazer backup de um pacote personalizado.
- É mais fácil distribuir pacotes instalados por você mesmo em uma rede (desde que a arquitetura seja consistente e o local da instalação seja relativamente isolado).

Eis as desvantagens:

- Exige tempo.
- Pacotes personalizados não se atualizam automaticamente. As distribuições mantêm a maioria dos pacotes atualizada, sem exigir muito trabalho. Essa é uma preocupação particular para pacotes que interajam com a rede, pois você deve garantir que sempre terá as atualizações de segurança mais recentes.
- Se você não usar realmente o pacote, estará desperdiçando seu tempo.
- Há uma chance de configurar indevidamente os pacotes.

Não faz muito sentido instalar pacotes como aqueles que estão no pacote `coreutils` que geramos anteriormente neste capítulo (`ls`, `cat` e assim por diante), a menos que você esteja criando um sistema bem personalizado. Por outro lado, se você tiver um interesse vital em servidores de rede como o Apache, a melhor maneira de ter um controle completo será instalar os servidores por conta própria.

16.4.1 Onde instalar

O prefixo default no GNU `autoconf` e em vários outros pacotes é `/usr/local` – o diretório tradicional para softwares instalados localmente. Os upgrades do sistema operacional ignoram `/usr/local`, portanto você não irá perder nada que esteja instalado ali durante uma atualização do sistema operacional; para instalações de softwares locais e pequenos, não haverá problemas com `/usr/local`. O único problema é que se você tiver muitos softwares personalizados instalados, isso poderá se transformar em uma verdadeira confusão. Milhares de pequenos arquivos incomuns poderão entrar na hierarquia de `/usr/local`, e você poderá não ter a mínima ideia da proveniência desses arquivos.

Se a situação realmente começar a sair de controle, crie seus próprios pacotes conforme descrito na seção 16.3.2.

16.5 Aplicando um patch

A maioria das alterações em códigos-fonte de software está disponível na forma de branches da versão online do código-fonte dos desenvolvedores (por exemplo, em um repositório git). Entretanto, ocasionalmente, você poderá obter um *patch* (correção) que deverá ser aplicado no código-fonte para corrigir bugs ou acrescentar funcionalidades. Você também poderá ver o termo *diff* usado como sinônimo para patch, pois o programa *diff* gera o patch.

O início de um patch tem a seguinte aparência:

```
--- src/file.c.orig    2015-07-17 14:29:12.000000000 +0100
+++ src/file.c        2015-09-18 10:22:17.000000000 +0100
@@ -2,16 +2,12 @@
```

Normalmente, os patches contêm alterações em mais de um arquivo. Procure três hifens seguidos (---) no patch para ver os arquivos que têm alterações e sempre observe o início de um patch para determinar o diretório de trabalho exigido. Note que o exemplo anterior faz referência a *src/file.c*. Sendo assim, você deve acessar o diretório que contém *src* antes de aplicar o patch, e não o diretório *src* propriamente dito.

Para aplicar o patch, execute o comando `patch`:

```
$ patch -p0 < arquivo_de_patch
```

Se tudo correr bem, `patch` terminará sem criar caso, deixando você com um conjunto atualizado de arquivos. Entretanto o `patch` poderá fazer a seguinte pergunta:

File to patch:

Geralmente, isso significa que você não está no diretório correto, porém pode indicar também que seu código-fonte não corresponde ao código-fonte no patch. Nesse caso, provavelmente você não está com sorte: mesmo que consiga identificar alguns dos arquivos para aplicar o patch, outros não serão adequadamente atualizados, deixando você com um código-fonte que não será possível compilar.

Em alguns casos, você poderá se deparar com um patch que se refira a uma versão de pacote, como em:

```
--- package-3.42/src/file.c.orig    2015-07-17 14:29:12.000000000 +0100
+++ package-3.42/src/file.c        2015-09-18 10:22:17.000000000 +0100
```

Se você tiver um número de versão um pouco diferente (ou tiver simplesmente renomeado o diretório), poderá dizer ao `patch` para remover os componentes iniciais do path. Por exemplo, suponha que você esteja no diretório que contém *src* (como antes). Para dizer ao `patch` para ignorar a parte referente a *package-3.42/* do path (ou seja, para remover um componente inicial do path), utilize `-p1`:

```
$ patch -p1 < arquivo_de_patch
```

16.6 Resolvendo problemas de compilação e de instalações

Se você compreende a diferença entre erros de compilador, avisos do compilador, erros de linker e problemas de bibliotecas compartilhadas, conforme descritos no capítulo 15, você não deverá ter muitas complicações para corrigir vários dos problemas que surgirem na geração de um software. Esta seção discute alguns problemas comuns. Embora seja pouco provável que você vá se deparar com qualquer um desses problemas ao gerar o software usando autoconf, não faz mal algum conhecer a aparência desses tipos de problema.

Antes de discutir as especificidades, certifique-se de que você possa ler determinados tipos de saída do `make`. É importante saber a diferença entre um erro e um erro ignorado. A seguir, apresentamos um erro verdadeiro, que deverá ser investigado:

```
make: *** [alvo] Error 1
```

Entretanto alguns Makefiles suspeitam que uma condição de erro possa ocorrer, porém sabem que esses erros são inofensivos. Normalmente, qualquer mensagem do tipo a seguir pode ser ignorada:

```
make: *** [alvo] Error 1 (ignored)
```

Além do mais, o GNU `make` geralmente chama a si mesmo várias vezes em pacotes grandes, com cada instância de `make` na mensagem de erro marcada com `[N]`, em que `N` é um número. Com frequência, você poderá encontrar o erro observando o erro do `make` que vem *imediatamente* após a mensagem de erro do compilador. Por exemplo:

```
[mensagem de erro do compilador envolvendo file.c]
make[3]: *** [file.o] Error 1
make[3]: Leaving directory '/home/src/package-5.0/src'
make[2]: *** [all] Error 2
make[2]: Leaving directory '/home/src/package-5.0/src'
make[1]: *** [all-recursive] Error 1
make[1]: Leaving directory '/home/src/package-5.0/'
make: *** [all] Error 2
```

As três primeiras linhas praticamente revelam tudo: o problema está centrado em torno de `file.c` localizado em `/home/src/package-5.0/src`. Infelizmente, há tantas informações extras de saída que pode ser difícil identificar os detalhes importantes. Saber como filtrar os erros subsequentes de `make` é muito importante para determinar a verdadeira causa do problema.

16.6.1 Erros específicos

A seguir, apresentamos alguns erros comuns de build que você poderá encontrar.

Problema

Mensagem de erro do compilador:

```
src.c:22: conflicting types for 'item'
/usr/include/file.h:47: previous declaration of 'item'
```

Explicação e correção

O programador fez uma nova declaração errônea de *item* na linha 22 de *src.c*. Geralmente, isso pode ser corrigido com a remoção da linha ofensora (com um comentário, um `#ifdef` ou o que quer que funcione).

Problema

Mensagem de erro do compilador:

```
src.c:37: 'time_t' undeclared (first use this function)
--trecho omitido--
src.c:37: parse error before '...'
```

Explicação e correção

O programador esqueceu-se de um arquivo crucial de cabeçalho. As páginas de manual são a melhor maneira de encontrar o arquivo de cabeçalho que está faltando. Inicialmente, dê uma olhada na linha ofensora (nesse caso, a linha 37 em *src.c*). Provavelmente, é uma declaração de variável como a que se segue:

```
time_t v1;
```

Procure `v1` no programa em busca de seu uso em torno de uma chamada de função. Por exemplo:

```
v1 = time(NULL);
```

Agora execute `man 2 time` ou `man 3 time` para procurar chamadas de sistema e de biblioteca de nome `time()`. Nesse caso, a página de manual da seção 2 contém o que precisamos:

SYNOPSIS

```
#include <time.h>
time_t time(time_t *t);
```

Isso significa que `time()` exige *time.h*. Coloque `#include <time.h>` no início de *src.c* e tente novamente.

Problema

Mensagem de erro do compilador (pré-processador):

```
src.c:4: pkg.h: No such file or directory  
(segue-se uma longa lista de erros)
```

Explicação e correção

O compilador executou o pré-processador C em *src.c*, mas não pôde encontrar o arquivo de inclusão *pkg.h*. O código-fonte provavelmente depende de uma biblioteca que precisa ser instalada, ou talvez você simplesmente deva fornecer o path de inclusão não padrão ao compilador. Geralmente, você só precisará adicionar uma opção `-I` de path de inclusão às flags do pré-processador C (`CPPFLAGS`). (Tenha em mente que também poderá ser necessário ter uma flag `-L` de linker para acompanhar os arquivos de inclusão.)

Se não parecer que uma biblioteca esteja faltando, há uma chance de você estar tentando uma compilação para um sistema operacional que não seja suportado por esse código-fonte. Confira o Makefile e os arquivos *README* para obter detalhes sobre as plataformas.

Se você estiver executando uma distribuição baseada em Debian, experimente executar o comando `apt-file` no nome do arquivo de cabeçalho:

```
$ apt-file search pkg.h
```

Isso poderá fazer com que o pacote de desenvolvimento de que você precisa seja encontrado. Para distribuições que disponibilizem o `yum`, você pode experimentar o seguinte:

```
$ yum provides */pkg.h
```

Problema

mensagem de erro do make:

```
make: prog: Command not found
```

Explicação e correção

Para gerar o pacote, é preciso ter *prog* em seu sistema. Se *prog* for algo como `cc`, `gcc` ou `ld`, você não terá os utilitários de desenvolvimento instalados em seu sistema. Por outro lado, se você acha que *prog* já está instalado em seu sistema, tente alterar o Makefile para especificar o nome do path completo de *prog*.

Em ocasiões raras, `make` gera *prog* e então o usa imediatamente, supondo que o diretório corrente (`.`) esteja em seu path de comandos. Se seu `$PATH` não incluir o

diretório corrente, você poderá editar o Makefile e mudar *prog* para *.prog*. De modo alternativo, você pode concatenar *.* ao seu path temporariamente.

16.7 Próximos passos

Discutimos somente o básico da geração de software. A seguir, apresentamos alguns tópicos adicionais que você poderá explorar após ter domínio sobre seus próprios builds:

- Entender como usar sistemas de build além do autoconf, por exemplo, o CMake e o SCons.
- Configurar builds para seus próprios softwares. Se você estiver criando seu próprio software, vai querer selecionar um sistema de build e aprender a usá-lo. Para empacotamentos com o GNU autoconf, o livro *Autotools* de John Calcote (No Starch Press, 2010) pode ajudar você.
- Compilar o kernel do Linux. O sistema de build do kernel é totalmente diferente do sistema de outras ferramentas. Ele tem seu próprio sistema de configuração para personalizar o kernel e os módulos. Contudo o procedimento é simples e, se você entender como o boot loader funciona, não deverá haver nenhum problema com ele. No entanto, tome cuidado ao fazer isso; certifique-se de sempre manter seu kernel antigo à mão caso não seja possível fazer o boot com o novo.
- Pacotes de código-fonte específicos de distribuição. As distribuições Linux mantêm suas próprias versões de código-fonte dos softwares na forma de pacotes especiais de código-fonte. Às vezes, você poderá encontrar patches úteis que expandam funcionalidades ou que corrijam problemas em pacotes que não tenham manutenção. Os sistemas de gerenciamento de pacotes de códigos-fonte incluem ferramentas para builds automáticos, por exemplo, o *debuild* do Debian e o *mock*, baseado em RPM.

Gerar um software geralmente é um passo importante para aprender sobre programação e desenvolvimento de software. As ferramentas que vimos nos dois últimos capítulos esclarecem o mistério acerca da procedência do software de seu sistema. Não é difícil dar os próximos passos, que consistem em olhar os códigos-fonte, fazer alterações e criar seu próprio software.

CAPÍTULO 17

Desenvolvendo sobre o básico

Os capítulos deste livro discutiram os componentes fundamentais de um sistema Linux, desde o kernel de baixo nível e a organização dos processos até a rede e outras ferramentas usadas para a geração de softwares. Com tudo isso esclarecido, o que você pode fazer agora? Muita coisa, na verdade! Pelo fato de o Linux suportar quase todo tipo de ambiente de programação não proprietário, é natural que uma variedade de aplicações esteja disponível. Vamos dar uma olhada em algumas áreas de aplicações em que o Linux é excelente e ver de que modo o que você aprendeu neste livro se relaciona com essas áreas.

17.1 Servidores e aplicações web

O Linux é um sistema operacional popular para servidores web, e o monarca reinante entre os servidores Linux é o Apache HTTP Server (geralmente referenciado somente como “Apache”). Outro servidor web sobre o qual você ouvirá com frequência é o Tomcat (também é um projeto Apache), que provê suporte para aplicações baseadas em Java.

Por si sós, os servidores web não fazem muito – eles podem servir arquivos, mas é só. O objetivo final da maioria dos servidores web como o Apache é prover uma plataforma subjacente para servir aplicações web. Por exemplo, a Wikipedia foi desenvolvida sobre o pacote MediaWiki, que você pode usar para criar a sua própria wiki. Os sistemas de gerenciamento de conteúdo como o Wordpress e o Drupal permitem que você crie seus próprios blogs e sites de mídia. Todas essas aplicações foram desenvolvidas em linguagens de programação que executam particularmente muito bem no Linux. Por exemplo, o MediaWiki, o Wordpress e o Drupal foram todos escritos em PHP.

Os blocos de construção que compõem as aplicações web são altamente modulares, portanto é fácil adicionar suas próprias extensões e criar aplicações usando frameworks como Django, Flask e Rails, que oferecem recursos para proporcionar infraestrutura e funcionalidades web comuns, por exemplo, templates, possibilidade de ter vários usuários e suporte a banco de dados.

Um servidor web que funcione bem depende de uma fundação sólida do sistema operacional. Em particular, o material nos capítulos de 8 a 10 é particularmente importante. Sua configuração de rede não deve ter nenhuma falha, mas talvez o mais importante seja entender o gerenciamento de recursos. Memória e disco dimensionados de forma adequada são fundamentais, especialmente se você planeja usar um banco de dados em sua aplicação.

17.2 Bancos de dados

Os bancos de dados são serviços especializados para armazenar e acessar dados, e muitos servidores de banco de dados e sistemas diferentes executam no Linux. Dois recursos principais dos bancos de dados os tornam atraentes: eles oferecem maneiras simples e uniformes de gerenciar dados individuais e em grupo e têm desempenho superior quanto ao acesso.

Os bancos de dados facilitam às aplicações analisar e alterar dados, especialmente quando comparados com o parsing e a alteração de arquivos-texto. Por exemplo, os arquivos */etc/passwd* e */etc/shadow* de um sistema Linux podem ser difíceis de manter em uma rede de computadores. De modo alternativo, você pode configurar um banco de dados que disponibilize informações de usuários por meio de LDAP (Lightweight Directory Access Protocol, ou Protocolo leve de acesso a diretórios) para fornecer essas informações ao sistema de autenticação do Linux. A configuração do lado cliente do Linux é simples; tudo o que você precisa fazer é editar o arquivo */etc/nsswitch.conf* e acrescentar algumas configurações extras.

A principal razão pela qual os bancos de dados geralmente oferecem um desempenho superior no acesso aos dados está no fato de usarem a indexação para monitorar a localização dos dados. Por exemplo, suponha que você tenha um conjunto de dados que represente uma lista contendo o primeiro nome e o sobrenome, além dos números de telefone. Um banco de dados pode ser usado para colocar um índice em qualquer um desses atributos, por exemplo, no sobrenome. Então, quando você estiver procurando uma pessoa com base no sobrenome, o banco de dados simplesmente consultará o índice relacionado ao sobrenome, em vez de pesquisar toda a lista.

17.2.1 Tipos de banco de dados

Os bancos de dados se apresentam em dois formatos básicos: relacional e não relacional. Os *bancos de dados relacionais*, também chamados de RDBMS (Relational Database Management Systems, ou Sistemas de gerenciamento de bancos de dados

relacionais), como o MySQL, o PostgreSQL, o Oracle e o MariaDB, são bancos de dados de propósito geral, excelentes para relacionar diferentes conjuntos de dados. Por exemplo, suponha que você tenha dois conjuntos de dados, um com CEPs e nomes e outro com os códigos postais e seus estados correspondentes. Um banco de dados relacional permite obter todos os nomes localizados em um determinado estado muito rapidamente. Normalmente, conversamos com bancos de dados relacionais usando uma linguagem de programação chamada SQL (Structured Query Language).

Os *bancos de dados não relacionais*, às vezes conhecidos como bancos de dados NoSQL, tendem a solucionar problemas particulares que não são facilmente tratados pelos bancos de dados relacionais. Por exemplo, bancos de dados para armazenamento de documentos, como o MongoDB, tentam fazer com que o armazenamento e a indexação de documentos inteiros sejam mais simples. Bancos de dados com chave-valor, como o redis, tendem a focar no desempenho. Bancos de dados NoSQL não têm uma linguagem de consulta comum como o SQL para fazer acessos. Em vez disso, você conversará com eles usando uma variedade de interfaces e de comandos.

Os problemas de desempenho de disco e de memória discutidos no capítulo 8 são extremamente importantes para a maioria das implementações de bancos de dados, pois há um compromisso entre a quantidade de dados que podemos armazenar em RAM (que é rápida) *versus* em disco. A maioria dos sistemas maiores de banco de dados também envolve um uso significativo de rede, pois esses sistemas estão distribuídos em vários servidores. A configuração mais comum de uma rede como essa recebe o nome de *replicação*, em que um banco de dados basicamente é copiado para vários servidores de banco de dados para aumentar o número de clientes que se conectam aos servidores.

17.3 Virtualização

Na maioria das empresas de grande porte, não é eficiente ter um hardware dedicado para realizar tarefas específicas de servidor, pois instalar um sistema operacional personalizado para uma tarefa em um servidor significa que você estará limitado a essa tarefa até que o sistema seja reinstalado. A tecnologia de máquinas virtuais possibilita instalar um ou mais sistemas operacionais simultaneamente (normalmente chamados de *convidados*, ou guests) em um único hardware e, em seguida, ativar e desativar os sistemas à vontade. Você pode até mesmo mover e copiar as máquinas virtuais para outros computadores.

Há vários sistemas de virtualização para Linux, por exemplo, o KVM (Kernel Virtual Machine) do kernel e o Xen. As máquinas virtuais são práticas, em especial, para

servidores web e de banco de dados. Embora seja possível configurar um único servidor Apache para servir vários sites, isso tem um custo no tocante à flexibilidade e à manutenção. Se esses sites forem administrados por diferentes usuários, você deverá gerenciar os servidores e os usuários ao mesmo tempo. Em vez disso, normalmente é preferível criar máquinas virtuais em um servidor físico com os próprios usuários suportados, para que eles não interfiram uns com os outros e você possa alterá-los e movê-los à vontade.

O software que opera as máquinas virtuais chama-se *hipervisor* (hypervisor). O hipervisor manipula várias partes dos níveis mais baixos de um sistema Linux que vimos neste livro; o resultado disso é que se você instalar um convidado Linux em uma máquina virtual, ele deverá se comportar exatamente como qualquer outro sistema Linux instalado.

17.4 Processamento distribuído e por demanda

Para facilitar o gerenciamento de recursos locais, você pode criar ferramentas sofisticadas sobre a tecnologia de máquinas virtuais. O termo *processamento na nuvem* (cloud computing) é um termo bem abrangente, geralmente usado como rótulo para essa área. Mais especificamente, o *IaaS* (Infrastructure as a Service, ou Infraestrutura como serviço) refere-se a sistemas que permitem prover e controlar recursos básicos de processamento como CPU, memória, armazenamento e rede em um servidor remoto. O projeto OpenStack é uma dessas APIs e plataformas que incluem o IaaS.

Indo além da infraestrutura básica, você também pode oferecer recursos de plataforma como sistema operacional, servidores de banco de dados e servidores web. Os sistemas que oferecem recursos nesse nível são chamados de *PaaS* (Platform as a Service, ou Plataforma como serviço).

O Linux é essencial para muitos desses serviços de processamento, pois, com frequência, é o sistema operacional subjacente a tudo isso. Quase todos os elementos que vimos neste livro, a começar pelo kernel, estão refletidos nesses sistemas.

17.5 Sistemas embarcados

Um *sistema embarcado* (embedded system) é algo projetado para servir a um propósito específico, por exemplo, um player de música, um streamer de vídeo ou um termostato. Compare isso com um desktop ou um sistema servidor, que podem lidar com vários tipos diferentes de tarefa (porém podem não fazer uma tarefa específica muito bem).

Podemos pensar nos sistemas embarcados quase como o oposto do processamento distribuído; em vez de expandir a escala do sistema operacional, um sistema embarcado geralmente (mas nem sempre) o reduz, com frequência, em um dispositivo pequeno. O Android talvez seja a versão embarcada mais popular do Linux em uso atualmente.

Os sistemas embarcados normalmente combinam hardware e software especializados. Por exemplo, podemos configurar um PC para que ele faça tudo o que um roteador wireless faz ao adicionar hardware de rede suficiente e configurar corretamente uma instalação Linux. Porém, em geral, é preferível comprar um dispositivo menor e dedicado, composto do hardware exigido, e eliminar qualquer hardware que não seja necessário. Por exemplo, um roteador precisa de mais portas de rede que a maioria dos desktops, porém não precisa de hardware para vídeo nem para áudio. E, depois que você tiver um hardware personalizado, você deverá personalizar o software do sistema, por exemplo, a parte interna do sistema operacional e a interface de usuário. O OpenWRT, mencionado no capítulo 9, é uma dessas distribuições Linux personalizadas.

O interesse pelos sistemas embarcados está aumentando à medida que mais hardwares pequenos e eficazes são introduzidos, particularmente, os projetos de SoC (System-on-a-Chip) que podem reunir um processador, memória e interfaces periféricas em um espaço reduzido. Por exemplo, os computadores Raspberry Pi e BeagleBone de uma só placa são baseados em um design desse tipo, com diversas variantes de Linux entre as quais podemos escolher para serem usadas como sistema operacional. Esses dispositivos têm uma saída de acesso fácil e dados de entrada provenientes de sensores que se conectam a interfaces de linguagem como o Python, tornando-os populares para prototipagem e dispositivos de pequeno porte.

As versões embarcadas de Linux variam no que diz respeito à quantidade de recursos em relação à versão servidor/desktop. Dispositivos pequenos e bem limitados devem remover tudo, exceto o mínimo necessário, por causa da falta de espaço que, geralmente, significa que até mesmo o shell e os utilitários essenciais se apresentam na forma de um único executável BusyBox. Esses sistemas tendem a exibir as maiores diferenças em relação a uma instalação Linux completa, e, com frequência, você verá softwares mais antigos neles, como o System V init.

Normalmente, você desenvolverá softwares para dispositivos embarcados usando um computador desktop comum. Os dispositivos mais eficazes, por exemplo, o Raspberry Pi, se dão ao luxo de ter mais área de armazenamento e a capacidade de executar softwares mais novos e mais completos, de modo que você poderá até mesmo executar várias ferramentas de desenvolvimento neles, de forma nativa.

Independentemente das diferenças, porém, os dispositivos embarcados continuam

compartilhando os genes do Linux descritos neste livro: você verá um kernel, um conjunto de dispositivos, interfaces de rede e um init, juntamente com um conjunto de processos de usuário. Os kernels embarcados tendem a ser próximos (ou idênticos) às versões normais de kernel, simplesmente com vários recursos desabilitados. Porém, à medida que você avançar em direção ao espaço de usuário, as diferenças se tornarão mais evidentes.

17.6 Observações finais

Independentemente de quais sejam seus objetivos ao entender melhor os sistemas Linux, espero que você tenha achado este livro útil. Meu objetivo foi incutir-lhe confiança quando você precisar entrar em seu sistema para fazer alterações ou algo novo. A essa altura, você deverá sentir que realmente tem controle sobre o seu sistema. Agora vá, force um pouco seus limites e divirta-se.

Bibliografia

- Abrahams, Paul W. e Bruce Larson, *UNIX for the Impatient*, 2ª ed. Boston: Addison-Wesley Professional, 1995.
- Aho, Alfred V., Brian W. Kernighan e Peter J. Weinberger, *The AWK Programming Language*. Boston: Addison-Wesley, 1988.
- Aho, Alfred V., Monica S. Lam, Ravi Sethi e Jeffrey D. Ullman, *Compiladores: princípios, técnicas e ferramentas*, 2ª ed. Pearson, 2008.
- Barrett, Daniel J., Richard E. Silverman e Robert G. Byrnes, *SSH, The Secure Shell: The Definitive Guide*, 2ª ed. Sebastopol: O'Reilly, 2005.
- Beazley, David M., *Python Essential Reference*, 4ª ed. Boston: Addison-Wesley, 2009.
- Beazley, David M., Brian D. Ward e Ian R. Cooke, “The Inside Story on Shared Libraries and Dynamic Loading”. *Computing in Science & Engineering* 3, nº 5 (setembro/outubro de 2001): 90-97.
- Calcote, John, *Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool*. São Francisco: No Starch Press, 2010.
- Carter, Gerald, Jay Ts e Robert Eckstein, *Using Samba: A File and Print Server for Linux, Unix & Mac OS X*, 3ª ed. Sebastopol: O'Reilly, 2007.
- Christiansen, Tom, brian d foy, Larry Wall e Jon Orwant, *Programming Perl: Unmatched Power for Processing and Scripting*, 4ª ed. Sebastopol: O'Reilly, 2012.
- Chromatic, *Modern Perl*, ed. rev. Hillsboro: Onyx Neon Press, 2014.
- Davies, Joshua, *Implementing SSL/TLS Using Cryptography and PKI*. Hoboken: Wiley, 2011.
- Filesystem Hierarchy Standard Group, “Filesystem Hierarchy Standard, Version 2.3”, editado por Rusty Russell, Daniel Quinlan e Christopher Yeoh, 2004, <http://www.pathname.com/fhs/>.
- Friedl, Jeffrey E. F., *Dominando expressões regulares: entenda os seus dados e seja mais produtivo*, 3ª ed. Rio de Janeiro: Alta Books, 2009.
- Gregg, Brendan, *Systems Performance: Enterprise and the Cloud*. Upper Saddle River: Prentice Hall, 2013.

- Grune, Dick, Kees van Reeuwijk, Henri E. Bal, Criel J.H. Jacobs e Koen Langendoen, *Modern Compiler Design*, 2ª ed. Nova York: Springer, 2012.
- Hopcroft, John E., Rajeev Motwani e Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation*, 3ª ed. Upper Saddle River: Prentice Hall, 2006.
- Kernighan, Brian W. e Rob Pike, *The UNIX Programming Environment*. Upper Saddle River: Prentice Hall, 1983.
- Kernighan, Brian W. e Dennis M. Ritchie, *C: a linguagem de programação – padrão Ansi*, Editora Campus, 1989.
- Kochan, Stephen G. e Patrick Wood, *Unix Shell Programming*, 3ª ed. Indianápolis: SAMS Publishing, 2003.
- Levine, John R., *Linkers and Loaders*. São Francisco: Morgan Kaufmann, 1999.
- Liu, Cricket, e Paul Albitz, *DNS and BIND*, 5ª ed. Sebastopol: O'Reilly, 2006.
- Lucas, Michael W., *SSH Mastery: OpenSSH, PuTTY, Tunnels and Keys*. Tilted Windmill Press, 2012.
- Matloff, Norman, *The Art of R Programming: A Tour of Statistical Software Design*. São Francisco: No Starch Press, 2011.
- Matloff, Norman e Peter Jay Salzman, *The Art of Debugging with GDB, DDD, and Eclipse*. São Francisco: No Starch Press, 2008.
- Mecklenburg, Robert, *Managing Projects with GNU Make: The Power of GNU Make for Building Anything*, 3ª ed. Sebastopol: O'Reilly, 2004.
- The New Hacker's Dictionary*, 3ª ed. Editado por Eric S. Raymond. Cambridge: MIT Press, 1996.
- Peek, Jerry, Grace Todino-Gonguet e John Strang, *Learning The UNIX Operating System: A Concise Guide for the New User*, 5ª ed. Sebastopol: O'Reilly, 2001.
- Pike, Rob, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey e Phil Winterbottom, "Plan 9 from Bell Labs." *Bell Labs*. Acessado em 10 de julho de 2014, <http://plan9.bell-labs.com/sys/doc/9.html>.
- Robbins, Arnold, *sed & awk Pocket Reference: Text Processing with Regular Expressions*, 2ª ed. Sebastopol: O'Reilly, 2002.
- Robbins, Arnold, Elbert Hannah e Linda Lamb, *Learning the vi and Vim Editors: Unix Text Processing*, 7ª ed. Sebastopol: O'Reilly, 2008.
- Salus, Peter H., *The Daemon, the Gnu, and the Penguin*. Reed Media Services, 2008.

- Samar, V. e R. Schemers, “Unified Login with Pluggable Authentication Modules (PAM)”, outubro de 1995, Open Software Foundation (RFC 86.0), <http://www.opengroup.org/rfc/mirror-rfc/rfc86.0.txt>.
- Schneier, Bruce, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2ª ed. Hoboken:Wiley, 1996.
- Shotts, William E. Jr., *The Linux Command Line: A Complete Introduction*. São Francisco: No Starch Press, 2012
- Schwartz, Randal L., brian d foy e Tom Phoenix, *Learning Perl: Making Easy Things Easy and Hard Things Possible*, 6ª ed. Sebastopol: O’Reilly, 2011.
- Silberschatz, Abraham, Peter B. Galvin e Greg Gagne, *Operating System Concepts*, 9ª ed. Hoboken: Wiley, 2012.
- Stallman, Richard M., *GNU Emacs Manual*, 17ª ed. Boston: Free Software Foundation, 2012. <http://www.gnu.org/software/emacs/manual/>.
- Stallman, Richard M., Roland Pesch, Stan Shebs, Etienne Suvasa e Matt Lee, *Debugging with GDB: The GNU Source-Level Debugger*, 10ª ed. Boston: GNU Press, 2011. <http://sourceware.org/gdb/current/onlinedocs/gdb/>.
- Stevens, W. Richard, *UNIX Network Programming, Volume 2: Interprocess Communications*, 2ª ed. Upper Saddle River: Prentice Hall, 1998.
- Stevens, W. Richard, Bill Fenner e Andrew M. Rudoff, *Unix Network Programming, Volume 1: The Sockets Networking API*, 3ª ed. Boston: Addison-Wesley Professional, 2003.
- Tanenbaum, Andrew S. e Herbert Bos, *Modern Operating Systems*, 4ª ed. Upper Saddle River: Prentice Hall, 2014.
- Tanenbaum, Andrew S. e David J. Wetherall, *Redes de computadores*, 5ª ed., Pearson, 2011